Programmer en langage C++

C. Delannoy

AVANT-PROPOS

La Programmation Orientée Objet (en abrégé P.O.O.) a pour ambition de faciliter l'activité de programmation, notamment en permettant de développer des "composants logiciels" réutilisables. Elle fait appel à des notions fondamentales (classes, objets, héritage, ligature dynamique ou polymorphisme...) inconnues de la plupart des langages de programmation traditionnels tels que C ou Pascal.

Un certain nombre de langages ont été définis de toutes pièces pour appliquer les concepts de P.O.O. ; citons Smalltalk, Simula, Eiffel... (on parle alors de "Langages Orientés Objet" ou L.O.O.). Le langage C++, quant à lui, a été conçu suivant une démarche quelque peu différente par B. Stroustrup (AT&T), à partir des années 1980 ; son objectif a été, en effet, d'adjoindre au langage C un certain nombre de spécificités lui permettant d'appliquer les concepts de P.O.O. Ainsi, C++ présente-t-il sur un "vrai L.O.O." l'originalité d'être fondé sur un langage répandu. Ceci laisse au programmeur toute liberté d'adopter un "style plus ou moins orienté objet", en se situant entre les deux extrêmes que constituent la poursuite d'une programmation classique d'une part, une pure P.O.O. d'autre part. Si une telle liberté présente le risque de céder, dans un premier temps, à la facilité en "mélangeant les genres" (la P.O.O. ne renie pas la programmation classique - elle l'enrichit), elle permet également une "transition en douceur" vers la P.O.O pure, avec tout le bénéfice qu'on peut en escompter à terme.

De sa conception jusqu'à sa normalisation, le langage C++ a quelque peu évolué. Plus précisément, un certain nombre de publications de AT&T ont servi de référence au cours des dernières années. Elles sont connues sous la terminologie suivante : version 1.1 en 1986, version 1.2 en 1987, version 2.0 en 1989, versions 2.1 et 3 en 1991 ; cette dernière a servi de base au travail du comité ANSI qui, sans la remettre en cause, l'a enrichie de quelques extensions et surtout de composants standard originaux se présentant sous forme de fonctions et de classes génériques.

Cet ouvrage est destiné à tous ceux qui souhaitent maîtriser le langage C++, que ce soit dans un but didactique ou en vue de développer de véritables applications. Conçu sous forme d'un cours, il suppose que vous possédez déjà quelques rudiments de C[1]. En effet, réaliser un livre de taille raisonnable ne nécessitant aucune connaissance préalable nous aurait immanquablement conduit à être succinct et, partant, à éluder les points qui se révèlent en réalité les plus fondamentaux lors du développement de logiciels en vraie grandeur.

Bien que nous adressant à un public "averti", nous avons cherché à rester pédagogique. Les différentes notions, en particulier celles relatives aux concepts de P.O.O, sont introduites progressivement. Les "références avant" ont toujours été évitées, de façon à permettre, le cas échéant, une étude séquentielle de

1 - Le cas échéant, on pourra trouver un cours complet de langage C dans Programmer en langage C du même auteur, chez le même éditeur.

l'ouvrage. Chaque notion est illustrée par un programme complet, assorti d'un exemple d'exécution montrant comment la mettre en œuvre dans un contexte réel. Celui-ci peut également servir :

- à une prise de connaissance intuitive ou à une révision rapide de la notion en question,
- à une expérimentation directe dans votre propre environnement de travail,
- de point de départ à une expérimentation personnelle.

Nous avons cherché à être complet non seulement en couvrant l'ensemble des possibilités du C++, mais également en approfondissant suffisamment certains aspects fondamentaux, de manière à rendre le lecteur parfaitement opérationnel dans la conception et le développement de ses propres classes. C'est ainsi que nous avons largement insisté sur le rôle du constructeur de recopie, ainsi que sur la redéfinition de l'opérateur d'affectation, éléments qui conduisent à la notion de "classe canonique". Toujours dans le même esprit, nous avons pris soin de bien développer les notions avancées mais indispensables que sont la ligature dynamique et les classes abstraites, lesquelles débouchent sur la notion la plus puissante du langage qu'est le polymorphisme. De même, nous avons largement insisté sur les notions de conteneur, d'itérateur et d'algorithme qui interviennent dans l'utisation de bon nombre de composants de la bibliothèque standard.

Notez que ce souci de complétude nous a amené à présenter des éléments d'intérêt limité (voire dangereux !) qui peuvent être ignorés dans un premier temps (notamment dans un cours de C++). Ils sont alors repérés par le pictogramme .

Les chapitres les plus importants ont été dotés d'exercices[2] comportant :

- des suggestions de manipulations destinées à mieux vous familiariser avec votre environnement ; par effet d'entraînement, elles vous feront probablement imaginer d'autres expérimentations de votre cru ;
- des programmes à rédiger ; dans ce cas, un exemple de correction est fourni en fin de volume.

Remarque concernant cette nouvelle édition

Les dernières éditions se basaient plutôt sur la version 3 et sur le projet de norme. Dorénavant, nous nous basons sur la norme elle-même, laquelle est reconnue de la plupart des compilateurs du marché, à quelques détails près (que nous signalons dans le texte). Toutefois, pour vous permettre d'utiliser d'anciens environnements de programmation, nous mentionnons les apports de la norme par rapport à la version 3 ainsi que les quelques différences avec les versions antérieures. Ces dernières ont surtout un caractère historique ; en particulier, elles mettent en avant les points délicats du langage pour lesquels la genèse a été quelque peu difficile.

Par ailleurs, cette nouvelle édition a été enrichie de quatre chapitres supplémentaires (XVII à XXI) et d'une annexe (F) décrivant les nouveaux composants standard, notamment les conteneurs et les algorithmes.

---

2 - De nombreux autres exercices peuvent être trouvés dans Exercices en langage C++ du même auteur, chez le même éditeur.

I. PRÉSENTATION GÉNÉRALE DE C++

Le langage C++ a été conçu (à partir de 1982) par Bjarne Stroustrup (AT&T Bell Laboratories) avec un objectif précis : ajouter au langage C des "classes" analogues à celles du langage Simula. Il s'agissait donc de "greffer" sur un langage classique des possibilités de "Programmation Orientée Objet". Avant de vous décrire le résultat auquel a abouti B. Stroustrup, nous commencerons par examiner succinctement ce qu'est la Programmation Orientée Objet d'une manière générale.

1. LA PROGRAMMATION ORIENTÉE OBJET

1.1 La problématique de la programmation

Jusqu'à maintenant, l'activité de programmation a toujours suscité des réactions diverses allant jusqu'à la contradiction totale. Pour certains, en effet, il ne s'agit que d'un jeu de construction enfantin, dans lequel il suffit d'enchaîner des instructions élémentaires (en nombre restreint) pour parvenir à résoudre (presque) n'importe quel problème. Pour d'autres au contraire, il s'agit de produire (au sens industriel du terme) des logiciels avec des exigences de qualité qu'on tente de mesurer suivant certains critères ; citons :

- l'exactitude : aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation (par exemple, données correspondant aux "spécifications"),
- la robustesse : aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation,
- l'extensibilité : facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications,
- la réutilisabilité : possibilité d'utiliser certaines parties ("modules") du logiciel pour résoudre un autre problème,
- la portabilité : facilité avec laquelle on peut exploiter un même logiciel dans différentes implémentations,
- l'efficience : temps d'exécution, taille mémoire...

La contradiction n'est souvent qu'apparente et essentiellement liée à l'importance des projets concernés. Par exemple, il est facile d'écrire un programme exact et robuste lorsqu'il comporte une centaine d'instructions ; il en va tout autrement lorsqu'il s'agit d'un projet de dix hommes-années ! De même, les aspects extensibilité et réutilisabilité n'auront guère d'importance dans le premier cas, alors qu'ils seront probablement cruciaux dans le second, ne serait-ce que pour des raisons économiques.

1.2 La programmation structurée

La programmation structurée a manifestement fait progresser la qualité de la production des logiciels. Mais, avec le recul, il faut bien reconnaître que ses propres fondements lui imposaient des limitations "naturelles". En effet, la programmation structurée reposait sur ce que l'on nomme souvent "l'équation de Wirth", à savoir :

Algorithmes + Structures de données = Programmes

Bien sûr, elle a permis de structurer les programmes, et, partant, d'en améliorer l'exactitude et la robustesse. On avait espéré qu'elle permettrait également d'en améliorer l'extensibilité et la réutilisabilité. Or, en pratique, on s'est aperçu que l'adaptation ou la réutilisation d'un logiciel conduisait souvent à "casser" le "module" intéressant, et ceci parce qu'il était nécessaire de remettre en cause une structure de données. Précisément, ce type de difficultés est l'émanation même de l'équation de Wirth qui découple totalement les données des procédures agissant sur ces données.

1.3 La Programmation Orientée Objet

a) Objet

C'est là qu'intervient la P.O.O. (abréviation de Programmation Orientée Objet), fondée justement sur le concept **d'objet**, à savoir une association des données et des procédures (qu'on appelle alors méthodes) agissant sur ces données. Par analogie avec l'équation de Wirth, on pourrait dire que l'équation de la P.O.O. est :

Méthodes + Données = Objet

b) Encapsulation

Mais cette association est plus qu'une simple juxtaposition. En effet, dans ce que l'on pourrait qualifier de P.O.O. "pure"[1], on réalise ce que l'on nomme une **encapsulation des données**. Cela signifie qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes qui jouent ainsi le rôle d'interface obligatoire. On traduit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un "message" à l'objet.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications[2] de ses méthodes, la manière dont sont réellement implantées les données étant sans importance. On décrit souvent une telle situation en disant qu'elle réalise une "abstraction des données" (ce qui exprime bien que les détails concrets d'implémentation sont cachés). A ce propos, on peut remarquer

---

1 - Nous verrons, en effet, que les concepts de la P.O.O. peuvent être appliqués d'une manière plus ou moins rigoureuse. En particulier, en C++, l'encapsulation ne sera pas obligatoire, ce qui ne veut pas dire qu'elle ne soit pas souhaitable.

2 - Noms, arguments et rôles.

qu'en programmation structurée, une procédure pouvait également être caractérisée (de l'extérieur) par ses spécifications, mais que, faute d'encapsulation, l'abstraction des données n'était pas réalisée.

L'encapsulation des données présente un intérêt manifeste en matière de qualité de logiciel. Elle facilite considérablement la maintenance : une modification éventuelle de la structure de données d'un objet n'a d'incidence que sur l'objet lui-même ; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification (ce qui n'était, bien sûr, pas le cas avec la programmation structurée). De la même manière, l'encapsulation des données facilite grandement la réutilisation d'un objet.

c) Classe

En P.O.O. apparaît généralement le concept de classe[3]. Ce dernier correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques. Une classe, en effet, n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune[4] et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O., on dit aussi qu'un objet est une "instance" de sa classe).

d) Héritage

Un autre concept important en P.O.O. est celui d'héritage. Il permet de définir une nouvelle classe à partir d'une classe existante (qu'on réutilise en bloc !), à laquelle on ajoute de nouvelles données et de nouvelles méthodes. La conception de la nouvelle classe, qui "hérite" des propriétés et des aptitudes de l'ancienne, peut ainsi s'appuyer sur des réalisations antérieures parfaitement au point et les "spécialiser" à volonté. Comme on peut s'en douter, l'héritage facilite largement la réutilisation de produits existants, et ceci d'autant plus qu'il peut être réitéré autant de fois que nécessaire (la classe C peut hériter de B, qui elle-même hérite de A)[5].

1.4 P.O.O. et langages

Nous venons d'énoncer les grands principes de la P.O.O., d'une manière générale, sans nous attacher à un langage particulier.

Or, manifestement, certains langages peuvent être conçus (de toutes pièces) pour appliquer à la lettre ces principes et réaliser ce que nous nommons de la P.O.O. "pure". C'est, par exemple, le cas de Simula, Smalltalk ou, plus récemment Eiffel. Le même phénomène a eu lieu, en son temps, pour la programmation structurée avec Pascal.

A l'opposé, on peut toujours tenter d'appliquer, avec plus ou moins de bonheur, ce que nous aurions tendance à nommer "une philosophie P.O.O." à un langage classique (Pascal, C...). On retrouve là une idée comparable à celle qui consistait à appliquer les principes de la programmation structurée à des langages comme Fortran ou Basic.

Le langage C++ se situe à mi-chemin entre ces deux points de vue. Il a, en effet, été obtenu en **ajoutant** à un langage classique (C) les outils permettant de mettre en oeuvre tous les principes de la P.O.O. Programmer en C++ va donc plus loin qu'adopter une philosophie P.O.O. en C, mais moins loin que de faire de la P.O.O. pure avec Eiffel !

---

3 - Dans certains langages (Turbo Pascal, par exemple), le mot classe est remplacé par objet et le mot objet par variable.

4 - Bien entendu, seule la structure est commune, les données étant propres à chaque objet. Les méthodes, par contre, sont effectivement communes à l'ensemble des objets d'une même classe.

5 - En C++, les techniques de méthodes virtuelles élargissent encore plus la portée de l'héritage ; mais il n'est pas possible, pour l'instant, d'en faire percevoir l'intérêt.

La solution adoptée par B. Stroustrup a le mérite de préserver l'existant (compatibilité avec C++ de programmes déjà écrits en C) ; elle permet également une "transition en douceur" de la programmation structurée vers la P.O.O. En revanche, elle n'impose nullement l'application stricte des principes de P.O.O. Comme vous le verrez, en C++, rien ne vous empêchera (sauf votre bon sens !) de faire cohabiter des objets (dignes de ce nom, parce que réalisant une parfaite encapsulation de leurs données), avec des fonctions classiques réalisant des effets de bord sur des variables globales...

2. C++, C ANSI ET P.O.O.

Précédemment, nous avons dit, d'une façon quelque peu simpliste, que C++ se présentait comme un "sur-ensemble" du langage C, offrant des possibilités de P.O.O. Il nous faut maintenant nuancer cette affirmation.

D'une part, lorsque l'on parle de sur-ensemble du C, nous nous référons au langage C tel qu'il est défini par la norme ANSI[6]. Or, en fait, il existe quelques incompatibilités entre le C ANSI et le C++. Celles-ci, comme nous le verrons, sont néanmoins relativement mineures ; elles sont, pour la plupart, dues à la différence "d'esprit" des deux langages, ainsi qu'à la tolérance dont a fait preuve la norme ANSI en cherchant à "préserver l'existant" (certaines tolérances ont disparu en C++).

D'autre part, les extensions du C++ par rapport au C ANSI ne sont pas toutes véritablement liées à la P.O.O. Certaines de ces extensions, en effet, pourraient être ajoutées avec profit au langage C, sans qu'il devienne pour autant "orienté objet"[7].

En fait, nous pourrions caractériser C++ par cette formule :

$$C++ = C \pm E + S + P$$

dans laquelle :

C désigne le C norme ANSI,

E représente les "écarts à la norme" de C++,

S représente les spécificités de C++ qui ne sont pas véritablement axées sur la P.O.O.,

P représente les possibilités de P.O.O.

3. LES INCOMPATIBILITÉS DE C++ AVEC LE C ANSI

Les principaux "écarts à la norme" sont décrits dans le chapitre II ; ils sont accompagnés de rappels concernant la norme C ANSI. Ils concernent essentiellement :

- les définitions de fonctions : en-têtes, prototypes, arguments et valeur de retour,
- la portée du qualificatif const,
- les compatibilités entre pointeurs.

---

6 - Laquelle a sensiblement évolué, par rapport à la définition initiale du langage C, réalisée en 1978, par Kernighan et Ritchie.

7 - D'ailleurs, certaines extensions de C++, par rapport à la première définition du C, ont été introduites dans le C ANSI (prototypes, fonctions à arguments variables...).

Il faut noter que ces incompatibilités sont restées les mêmes au fil des différentes versions de C++.

4. LES SPÉCIFICITÉS DE C++

Comme nous l'avons dit, C++ présente, par rapport au C ANSI, des extensions qui ne sont pas véritablement orientées P.O.O. Elles seront décrites dans le chapitre IV. En voici un bref résumé :

- nouvelle forme de commentaire (en fin de ligne),
- plus grande liberté dans l'emplacement des déclarations,
- notion de référence facilitant la mise en œuvre de la transmission d'arguments par adresse,
- surdéfinition des fonctions : attribution d'un même nom à différentes fonctions, la reconnaissance de la fonction réellement appelée se faisant d'après le type et le nombre des arguments figurant dans l'appel (on parle parfois de signature),
- nouveaux opérateurs de gestion dynamique de la mémoire : new et delete,
- possibilité de définir des fonctions "en ligne" (inline), ce qui accroît la vitesse d'exécution, sans perdre pour autant le formalisme des fonctions.

5. C++ ET LA PROGRAMMATION ORIENTÉE OBJET

Les possibilités de P.O.O. représentent, bien sûr, l'essentiel de l'apport de C++.

C++ dispose de la notion de classe (généralisation de la notion de type défini par l'utilisateur). Une classe comportera :

- la description d'une structure de données,
- des méthodes.

Sur le plan du vocabulaire, C++ utilise des termes qui lui sont propres. On parle, en effet :

- de "membres donnée" pour désigner les différents membres de la structure de données associée à une classe,
- de "fonctions membre" pour désigner les méthodes.

A partir d'une classe, on pourra "instancier" des objets (nous dirons généralement créer des objets) :

- soit par des déclarations usuelles (de type classe),
- soit par allocation dynamique, en faisant appel au nouvel opérateur new.

C++ permet l'encapsulation des données, mais il ne l'impose pas. On peut le regretter mais il ne faut pas perdre de vue que, de par sa conception même (extension de C), le C++ ne peut pas, de toute façon, être un langage de P.O.O. pure. Bien entendu, il reste toujours possible au concepteur de faire preuve de rigueur, en s'astreignant à certaines règles telles que l'encapsulation absolue.

C++ permet de définir ce que l'on nomme des "constructeurs" de classe. Un constructeur est une fonction membre particulière qui est exécutée au moment de la création d'un objet de la classe. Le constructeur peut notamment prendre en charge "l'initialisation d'un objet", au sens le plus large du terme, c'est-à-dire sa mise dans un état initial permettant son bon fonctionnement ultérieur ; il peut s'agir de banales initialisations de membres donnée, mais également d'une préparation plus élaborée correspondant au déroulement d'instructions, voire d'une allocation dynamique d'emplacements nécessaires à l'utilisation de l'objet.

L'existence d'un constructeur garantit que l'objet sera toujours initialisé, ce qui constitue manifestement une sécurité.

D'une manière similaire, une classe peut disposer d'un "destructeur", fonction membre exécutée au moment de la destruction d'un objet. Celle-ci présentera surtout un intérêt dans le cas d'objets effectuant des allocations dynamiques d'emplacements ; ces derniers pourront être libérés par le destructeur.

Une des originalités de C++, par rapport à d'autres langages de P.O.O., réside dans la possibilité de définir des "fonctions amies d'une classe". Il s'agit de fonctions "usuelles" (qui ne sont donc pas des fonctions membre d'une classe) qui sont autorisées (par une classe) à accéder aux données (encapsulées) de la classe. Certes, le principe d'encapsulation est violé, mais uniquement par des fonctions dûment autorisées à le faire.

La classe est un type défini par l'utilisateur. La notion de "surdéfinition d'opérateurs" va permettre de doter cette classe d'opérations analogues à celles que l'on rencontre pour les types prédéfinis. Par exemple, on pourra définir une classe complexe (destinée à représenter des nombres complexes) et la munir des opérations d'addition, de soustraction, de multiplication et de division. Qui plus est, ces opérations pourront utiliser les symboles existants : +, -, *, /.

C dispose de possibilités de conversions (explicites ou implicites). C++ permet d'élargir ces conversions aux types définis par l'utilisateur que sont les classes. Par exemple, on pourra donner un sens à la conversion int -> complexe ou à la conversion complexe -> float (complexe étant une classe).

Naturellement, C++ dispose de l'héritage et même (depuis la version 2.0) de possibilités dites "d'héritage multiple" permettant à une classe d'hériter simultanément de plusieurs autres.

En matière d'entrées-sorties, C++ comporte de nouvelles possibilités basées sur la notion de "flot". Leurs avantages sur les entrées-sorties de C sont, en particulier :

- simplicité d'utilisation,
- taille mémoire réduite (on n'introduit que ce qui est utile),
- possibilité de leur donner un sens pour les types définis par l'utilisateur que sont les classes (grâce au mécanisme de surdéfinition d'opérateur).

Bien qu'elles soient liées à l'aspect P.O.O., nous ferons une première présentation de ces nouvelles possibilités d'entrées-sorties dès le chapitre III. Ceci nous permettra de réaliser rapidement des programmes dans l'esprit du C++.

Dans ses dernières versions, le C++ a été doté de la notion de patron. Elle permet de définir des modèles utilisables pour générer différentes classes ou différentes fonctions qualifiées parfois de génériques, même si cette généricité n'est pas totalement intégrée dans le langage lui-même comme c'est par exemple le cas avec ADA.

Enfin, la norme ANSI a notablement accru le contenu de la bibliothèque standard de C++ qui vient compléter celle du C, toujours disponible. En particulier, on y trouve des nombreux patrons de classes et de fonctions permettant de mettre en œuvre les structures de données et les algorithmes les plus usuels, évitant ainsi d'avoir à réinventer la roue à la moindre occasion.

II. LES INCOMPATIBILITÉS DE C++ AVEC LE C ANSI

A priori, le langage C++ peut être considéré comme une extension du langage C, tel qu'il est défini par la norme ANSI. Tout programme écrit en C devrait donc pouvoir être traduit correctement par un compilateur C++ et son exécution devrait alors fournir les mêmes résultats que ceux obtenus en utilisant un compilateur C.

Ce point de vue correspond effectivement au souhait du concepteur du langage C++. Néanmoins, en pratique, un certain nombre "d'incompatibilités" avec le C ANSI ont subsisté, Elles sont essentiellement inhérentes à l'esprit même dans lequel les deux langages ont été conçus.

Nous allons décrire ici les incompatibilités les plus importantes en pratique, en particulier celles qui se révéleraient quasiment à coup sûr dans la mise au point de vos premiers programmes C++. A ce propos, notez que nous aurions pu nous contenter de "citer" ces incompatibilités. Néanmoins, celles-ci sont définies par rapport au C ANSI ; elles concernent, pour la plupart, des possibilités qui n'existaient pas dans la première définition de Kerninghan et Ritchie et qui ne sont donc pas (encore) reconnues de toutes les implémentations du C. Aussi, en profitons-nous pour vous exposer (ou rappeler) ce que sont ces possibilités du C ANSI, en même temps que nous vous montrons ce qu'elles sont devenues en C++.

Par ailleurs, quelques autres incompatibilités mineures seront abordées au fil des prochains chapitres ; elles seront toutes récapitulées dans l'annexe B.

1. LES DÉFINITIONS DE FONCTIONS EN C++

Suivant la norme ANSI, il existe en C deux façons de définir[1] une fonction. Supposez, par exemple, que nous ayons à définir une fonction nommée fexple, fournissant une valeur de retour[2] de type double et recevant deux arguments, l'un de type int, l'autre de type double. Nous pouvons, pour cela, procéder de l'une des deux façons suivantes :

```
double fexple (u, v)
int u ;
double v ;
```

```
double fexple (int u, double v)
{
    ...
```

1 - Ne confondez pas la "définition" d'une fonction qui correspond à la description, à l'aide d'instructions C, de "ce que fait" une fonction avec sa "déclaration" qui correspond à une simple information (nom de la fonction et, éventuellement, type des arguments et de la valeur de retour) fournie au compilateur.

2 - On parle également de "résultat fourni par la fonction", de "valeur retournée"...

```
{                                                    ...  /* corps de la fonction */
   ...  /* corps de la fonction */      }
}
```

La première forme était la seule prévue par la définition initiale de Kernighan et Ritchie. La seconde a été introduite par la norme ANSI qui n'a, toutefois, pas exclu l'ancienne[3].

Le langage C++ n'accepte, quant à lui, que la seconde forme :

```
double fexple (int u, double v)
{
   ...  /* corps de la fonction */
}
```

Remarque :

Comme en C ANSI, lorsqu'une fonction fournit une valeur de type int, le mot int peut être omis dans l'en-tête. Nous ne vous conseillons guère, toutefois, d'employer cette possibilité qui nuit à la lisibilité des programmes.

2. LES PROTOTYPES EN C++

Nous venons de voir que le C++ était plus restrictif que le C ANSI en matière de définition de fonctions. Il en va de même pour les déclarations de fonctions. En C ANSI, lorsque vous utilisiez une fonction qui n'avait pas été définie auparavant dans le même fichier source, vous pouviez :

- ne pas la déclarer (on considérait alors que sa valeur de retour était de type int),
- la déclarer en ne précisant que le type de la valeur de retour, par exemple :

```
double fexple ;
```

- la déclarer à l'aide de ce que l'on nomme un "prototype", par exemple :

```
double fexple (int, double) :
```

En C++, un appel de fonction ne sera accepté que si le compilateur connaît le type des arguments et celui de sa valeur de retour. Ce qui signifie que la fonction en question doit **avoir fait l'objet d'une déclaration sous forme d'un prototype** (ou, à la rigueur, avoir été préalablement définie dans le même fichier source[4]).

N'oubliez pas que, à chaque fois que le compilateur rencontre un appel de fonction, il compare les types des "arguments effectifs" avec ceux des arguments muets correspondants[5]. En cas de différence, il met en place les conversions nécessaires pour que la fonction reçoive des arguments du bon type. Les conversions possibles ne se limitent pas aux "conversions non dégradantes" (telles que, par exemple, char -> double, int -> long). En effet, elles comportent toutes les conversions autorisées lors d'une affectation. On peut donc y rencontrer des "conversions dégradantes" telles que, par exemple, int -> char, double -> float, double -> int, ...

Voici un exemple illustrant ce point :

---

3 - Dans le seul but de rendre compatible avec la norme, des anciens programmes ou des anciens compilateurs.

4 - Toutefois, même dans ce cas, le prototype reste conseillé, notamment pour éviter tout problème en cas d'éclatement du fichier source.

5 - Nous supposons que le compilateur connaît le type des arguments de la fonction, ce qui est toujours le cas en C++.

```
double fexple (int, double) ;      /* déclaration de fexple */
   .....
main()
{
   int n ;
   char c ;
   double z, res1, res2, res3 ;
     .....

   res1 = fexple (n, z) ;      /* appel "normal" - aucune conversion */
   res2 = fexple (c, z) ;      /* conversion, avant appel, de c en int */
   res3 = fexple (z, n) ;      /* conversion, avant appel, de z en int */
                               /*                         et de n en double */
     .....
}
```

Exemple de conversions de types lors de l'appel d'une fonction

Remarques

1) Lorsque la définition de la fonction et sa déclaration (sous forme d'un prototype) figurent dans le même fichier source, le compilateur est en mesure de vérifier la cohérence entre l'en-tête de la fonction et le prototype. S'il n'y a pas correspondance (exacte cette fois) de type, on obtient une erreur de compilation. Voici un exemple correct :

```
double fexple (int, double) ;    /* déclaration de fexple */
main()
{   .....
}
     /*   définition de fexple   */
double fexple (int u, double v)
{    /* corps de la fonction */
}
```

En revanche, celui-ci conduit à une erreur de compilation :

```
double fexple (int, float) ;    /* déclaration de fexple */
main()
{    .....
}
    /*   définition de fexple   */
double fexple (int u, double v)
{   /* corps de la fonction */
}
```

Bien entendu, si la définition de la fonction et sa déclaration (donc son utilisation[6]) ne figurent pas dans le même fichier source, aucun contrôle ne peut plus être effectué par le compilateur. En général,

6 - A moins qu'on ne l'ait déclarée, sans l'utiliser, ce qui arrive fréquemment lorsque l'on fait appel à des fichiers en-tête

à partir du moment où l'on doit utiliser une fonction en dehors du fichier où elle est définie (ou, à partir de plusieurs fichiers source différents), on place son prototype dans un fichier en-tête ; ce dernier est incorporé, en cas de besoin, par la directive #include, ce qui évite tout risque de faute d'écriture du prototype.

2) Comme en C, la "portée" du prototype est limitée à :

- la partie du fichier source située à la suite de sa déclaration, si elle figure à un niveau global, c'est-à-dire en dehors de toute définition de fonction[7] ; c'était le cas du prototype de fexple dans nos précédents exemples,
- la fonction dans laquelle il figure, dans le cas contraire.

3) De manière analogue à ce qui se produit pour l'en-tête d'une fonction, lorsqu'un prototype ne précise pas le type de la valeur de retour, comme dans :

```
bizarre (int, float, char) ;
```

C++ considérera qu'il s'agit, par défaut, du type int. Ainsi, le prototype précédent est-il équivalent à :

```
int bizarre (int, float, char) ;
```

Nous vous conseillons d'éviter cette possibilité qui nuit à la lisibilité des programmes.

4) Le prototype peut prendre une forme plus étoffée[8], dans laquelle figurent également des noms d'arguments. Ainsi, le prototype de notre fonction fexple du début du paragraphe 2 pourrait également s'écrire :

```
double fexple (int a, double x) ;
```

ou encore :

```
double fexple (int u, double v) ;
```

Dans ce cas, les noms d'arguments (a et x dans le premier exemple, u et v dans le second) ne jouent aucun rôle. Ils sont purement et simplement ignorés du compilateur, de sorte que ces prototypes restent parfaitement équivalents aux précédents. On peut trouver un intérêt à cette possibilité lorsque l'on souhaite accompagner ce prototype de commentaires décrivant le rôle des différents arguments (cela peut s'avérer pratique dans le cas où l'on place ce prototype dans un fichier en-tête).

5) Certains compilateurs semblent ne pas imposer l'emploi de prototypes. En réalité, ils fabriquent automatiquement un prototype à la rencontre du premier appel d'une fonction en tenant compte de ses arguments effectifs. Une telle démarche qui semble partir d'un bon sentiment (alléger la tâche de l'utilisateur) peut se révéler désastreuse. Actuellement, toutefois, la tendance est de n'effectuer ce "prototypage automatique" que sur demande explicite de l'utilisateur ou dans le cas de compilation de programmes écrits en C (pour éviter leur modification systématique).

---

7 - Y compris la fonction main.

8 - Nommée parfois prototype complet ; l'autre étant nommée prototype réduit.

3. ARGUMENTS ET VALEUR DE RETOUR D'UNE FONCTION

3.1 Points communs à C et C++

En C++, comme en C ANSI, les arguments d'une fonction ainsi que la valeur de retour peuvent :

- ne pas exister,
- être une valeur "scalaire", d'un des types de base (caractères, entiers, flottants, pointeurs),
- être une valeur d'un type structure.

La dernière possibilité a été introduite par la norme ANSI. Nous verrons qu'en C++, elle se généralise aux objets d'un type classe. Pour l'instant, notons simplement qu'il subsiste en C++, comme en C ANSI, une disparité entre les tableaux et les structures puisque :

- il est possible de transmettre la valeur d'une structure, aussi bien en argument qu'en valeur de retour,
- il n'est pas possible de faire de même avec les tableaux.

Notez qu'il s'agit là d'une disparité difficile à résorber, compte tenu de la volonté de rendre équivalents le nom d'un tableau et son adresse.

Bien entendu, il est toujours possible de transmettre l'adresse d'un tableau, mais cette remarque vaut également pour une structure.

3.2 Différences entre C et C++

En fait, les différences ne portent que sur la syntaxe des en-têtes et des prototypes des fonctions, et ceci uniquement dans deux cas précis :

- fonctions sans arguments,
- fonctions sans valeur de retour.

a) Fonctions sans arguments

Alors qu'en C ANSI on peut employer le mot void pour définir (en-tête) ou déclarer (prototype) une fonction sans argument, en C++, on fournit une "liste vide". Ainsi, là où en C, on déclarait :

```
float fct (void) ;
```

on déclarera, en C++ :

```
float fct ( ) ;
```

b) Fonctions sans valeur de retour

En C ANSI, on **peut** utiliser le mot void pour définir (en-tête) ou déclarer (prototype) une fonction sans valeur de retour. En C++, on **doit** absolument le faire, comme dans cet exemple :

```
void fct (int, double) ;
```

La déclaration :

```
fct (int, double) ;
```

conduirait C++ à considérer que fct fournit une valeur de retour de type int (voir la remarque du paragraphe 1).

4. LE QUALIFICATIF CONST

La norme C ANSI a introduit le qualificatif const. Il permet de spécifier qu'un symbole correspond à "quelque chose" dont la valeur ne doit pas changer, ce qui peut permettre au compilateur de signaler les tentatives de modification (lorsque cela lui est possible !).

Ceci reste vrai en C++. Cependant, un certain nombre de différences importantes apparaissent, au niveau de la portée du symbole concerné et de son utilisation dans une expression.

4.1 Portée

Lorsque const s'applique à des variables locales automatiques, aucune différence n'existe entre C et C++, la portée étant limitée au bloc ou à la fonction concernée par la déclaration.

En revanche, lorsque const s'applique à une variable globale, C++ limite la portée du symbole au fichier source contenant la déclaration (comme s'il avait reçu l'attribut static) ; C ne faisait aucune limitation.

Pourquoi cette différence ? La principale raison réside dans l'idée qu'avec la règle adoptée par C++ il devient plus facile de remplacer certaines instructions #define par des déclarations de constantes ; Ainsi, là où en C vous procédiez de cette façon :

```
#define N 8                          #define N 3
   .....                                .....

 fichier 1                            fichier 2
```

vous pouvez, en C++, procéder ainsi :

```
const int N = 8 ;                    const int N = 3 ;
   .....                                .....

 fichier 1                            fichier 2
```

En C, vous auriez obtenu une erreur au moment de l'édition de liens. Vous auriez pu l'éviter :

- soit en déclarant N static, dans au moins un des deux fichiers (ou, mieux, dans les deux) :

  ```
  static const int N = 8 ;                    static const int N = 3 ;
  ```

  Ceci aurait alors été parfaitement équivalent à ce que fait C++ avec les premières déclarations ;
- soit, si N avait eu la même valeur dans les deux fichiers, en plaçant, dans le second fichier :

  ```
  extern const int N ;
  ```

  mais, dans ce cas, il ne se serait plus agi d'un remplacement de #define.

4.2 Utilisation dans une expression

Rappelons que l'on nomme "expression constante" une expression dont la valeur est calculée lors de la compilation. Ainsi, avec :

```
const int p = 3 ;
```

l'expression :

```
2 * p * 5
```

n'est pas une expression constante en C alors qu'elle est une expression constante en C++.

Ce point est particulièrement sensible dans les déclarations de tableaux (statiques ou automatiques) dont les dimensions doivent obligatoirement être des expressions constantes (même pour les tableaux automatiques, le compilateur doit connaître la taille à réserver sur la pile !). Ainsi, les instructions :

```
const int nel = 15 ;
.....
double t1 [nel + 1], t2[2 * nel] [nel] ;
```

seront acceptées en C++, alors qu'elles étaient refusées en C.

Remarques :

1) En toute rigueur, la possibilité que nous venons de décrire ne constitue pas une incompatibilité entre C et C++ puisqu'il s'agit d'une facilité supplémentaire.

2) D'une manière générale, C++ a été conçu pour limiter au maximum l'emploi des directives du préprocesseur (on devrait pouvoir se contenter de #include et des directives d'inclusion conditionnelle). Les modifications apportées au qualificatif const vont effectivement dans ce sens[9].

5. COMPATIBILITÉ ENTRE LE TYPE VOID * ET LES AUTRES POINTEURS

En C ANSI, le "type générique" void * est compatible avec les autres types pointeurs, et ceci dans les deux sens. Ainsi, avec ces déclarations :

```
void * gen ;
int * adi ;
```

Ces deux affectations sont légales en C ANSI :

```
gen = adi ;
adi = gen ;
```

Elles font intervenir des "conversions implicites", à savoir :

9 - Encore faut-il que le programmeur C++ accepte de changer les habitudes qu'il avait dû prendre en C (faute de pouvoir faire autrement) !

int * -> void * pour la première,
void * -> int * pour la seconde.

En C++, seule la conversion d'un pointeur quelconque en void * peut être implicite. Ainsi, avec les déclarations précédentes, seule l'affectation

```
gen = adi ;
```

est acceptée. Bien entendu, il reste toujours possible de faire appel explicitement à la conversion void * -> int * en utilisant l'opérateur de "cast" :

```
adi = (int *) gen ;
```

Remarque :

On peut dire que la conversion d'un pointeur de type quelconque en void * revient à ne s'intéresser qu'à l'adresse correspondant au pointeur, en ignorant son type. La conversion inverse, en revanche, de void * en un pointeur de type donné, revient à associer (peut-être arbitrairement !) un type à une adresse. Manifestement, cette deuxième possibilité est plus dangereuse que la première ; elle peut même obliger le compilateur à introduire des modifications de l'adresse de départ, dans le seul but de respecter certaines contraintes d'alignement (liées au type d'arrivée). C'est la raison pour laquelle cette conversion ne fait plus partie des conversions implicites en C++.

III. LES NOUVELLES POSSIBILITÉS D'ENTRÉES-SORTIES CONVERSATIONNELLES DE C++

C++ dispose de toutes les routines offertes par la "bibliothèque standard" du C ANSI. Mais il comporte également de nouvelles possibilités d'entrées-sorties. Celles-ci reposent sur les notions de "flots" et de "surdéfinition d'opérateur" que nous n'aborderons qu'ultérieurement. Toutefois, il ne serait pas judicieux d'attendre que vous ayez étudié ces différents points pour commencer à écrire des programmes complets, rédigés dans l'esprit de C++. C'est pourquoi nous allons dans ce chapitre vous présenter, de manière assez informelle, ces nouvelles possibilités d'entrées-sorties en nous limitant cependant à ce que nous nommons l'aspect "conversationnel" (lecture sur "l'entrée standard", écriture sur la "sortie standard").

1. LES NOUVELLES ENTRÉES-SORTIES EN C++

Les "routines" (fonctions et macros) de la "bibliothèque standard" du C ANSI[1], donc, en particulier, celles relatives aux entrées-sorties, sont utilisables en C++ ; pour ce faire, il vous suffit d'inclure les fichiers en-tête habituels, pour obtenir les prototypes et autres déclarations nécessaires à leur bonne utilisation.

Mais C++ dispose en plus de possibilités d'entrées-sorties ayant les caractéristiques suivantes :

- simplicité d'utilisation : en particulier, on pourra souvent s'y affranchir de la notion de "format", si chère aux fonctions de la famille printf ou scanf,
- diminution de la taille du "module objet[2]" correspondant : alors que, par exemple, un seul appel de printf introduit obligatoirement, dans le module objet, un ensemble d'instructions en couvrant toutes les éventualités, l'emploi des nouvelles possibilités offertes par C++ n'amènera que les seules instructions nécessaires,
- possibilités extensibles aux types que vous définirez vous-même sous forme de classes.

Voyons donc, dès maintenant, comment utiliser ces possibilités dans le cas de l'entrée standard ou de la sortie standard.

---

1 - L'un des grands mérites de cette norme est de définir, outre le langage C lui-même, les caractéristiques d'un certain nombre de routines formant ce que l'on nomme la "bibliothèque standard".

2 - Ensemble d'instructions, en langage machine, résultant de la traduction d'un fichier source.

2. ÉCRITURE SUR LA SORTIE STANDARD

2.1 Quelques exemples

Avant d'étudier les différentes possibilités offertes par C++, examinons quelques exemples.

a) Exemple 1

Voyez ce premier cas, très simple, accompagné de son exemple d'exécution :

```
#include <iostream.h> /* indispensable pour utiliser cout
*/
                      /*   suivant l'implémentation : on peut utiliser
<iostream> */
                      /*   ou <iostream.hpp> <iostream.hxx> ...
*/
main()
{
   cout << "bonjour" ;
}
```

```
bonjour
```

Ecriture en C++ (1)

Notez, tout d'abord, que nous avons inclus un fichier en-tête nommé iostream.h ; son implémentation peut varier avec les implémentations de C++ ; dans l'avenir, on devrait plutôt rencontrer iostream, sans extension. Ce fichier contient toutes les déclarations nécessaires à l'utilisation des entrées-sorties spécifiques au C++.

L'interprétation détaillée de l'instruction :

```
cout << "bonjour" ;
```

nécessiterait des connaissances qui ne seront introduites qu'ultérieurement. Pour l'instant, il vous suffit d'admettre les points suivants :

- cout désigne un "flot de sortie" prédéfini, associé à l'entrée standard du C (stdout),
- << est un "opérateur" dont l'opérande de gauche (ici cout) est un flot et l'opérande de droite une expression de type quelconque. L'instruction précédente peut être interprétée comme ceci : le flot cout reçoit la valeur "bonjour".

b) Exemple 2

```
#include <iostream.h>
main()
{
   int n = 25 ;
   cout << "valeur : " ;
   cout << n ;
}
```

```
valeur : 25
```

Ecriture en C++ (2)

Il ressemble au précédent, mais, cette fois, vous constatez que nous avons utilisé le même opérateur << pour envoyer sur le flot cout, d'abord une information de type chaîne, ensuite une information de type entier. Le rôle de l'opérateur << est manifestement différent dans les deux cas : dans le premier, on a transmis les caractères de la chaîne, dans le second, on a procédé à un "formatage[3]" pour "convertir" une valeur binaire entière en une suite de caractères. Cette possibilité d'attribuer plusieurs significations à un même opérateur correspond à ce que l'on nomme en C++, la "surdéfinition d'opérateur" (que nous aborderons en détail dans le chapitre IX).

c) Exemple 3

Jusqu'ici, nous avons écrit une instruction différente pour chaque information transmise au flot cout. En fait, les deux instructions :

```
cout << "valeur :" ;
cout << n ;
```

peuvent se condenser en une seule :

```
cout << "valeur :" << n ;
```

Là encore, l'interprétation exacte de cette possibilité sera fournie ultérieurement mais, d'ores et déjà, nous pouvons dire qu'elle réside dans deux points :

- l'opérateur << est (comme l'opérateur "original") associatif de gauche à droite,
- le résultat fourni par l'opérateur <<, quand il reçoit un flot en premier opérande est ce même flot, après qu'il ait reçu l'information concernée.

Ainsi, l'instruction précédente est-elle équivalente à :

```
(cout << "valeur :") << n ;
```

Celle-ci peut s'interpréter comme ceci :

- dans un premier temps, le flot cout reçoit la chaîne "bonjour",
- dans un deuxième temps, le flot (cout << "bonjour"), c'est-à-dire le flot cout augmenté de "bonjour", reçoit la valeur de n.

3 - Ici, ce formatage est implicite ; dans le cas de printf, on l'aurait explicité (%d).

Notez bien que, si cette interprétation ne vous paraît pas évidente, il vous suffit d'admettre pour l'instant qu'une instruction telle que :

```
cout << ----- << ----- << ----- << ----- ;
```

permet d'envoyer sur le flot cout les informations symbolisées par des traits, dans l'ordre où elles apparaissent.

2.2 Les possibilités d'écriture sur cout

Nous venons de voir des exemples d'écriture de chaînes et d'entiers. D'une manière générale, vous pouvez utiliser l'opérateur << pour envoyer sur cout la valeur d'une expression de l'un des types suivants :

- type de base quelconque (caractère, entier signé ou non, flottant),
- chaîne de caractères (char *)[4] : on obtient l'affichage des caractères constituant la chaîne,
- pointeur, autre que char *[5] : on obtient l'adresse correspondante (en hexadécimal) ; si on veut obtenir l'adresse d'une chaîne de caractères et non les caractères qu'elle contient, on peut toujours convertir cette chaîne (de type char *) en void *.

Voici un exemple de programme illustrant ces possibilités d'écriture sur cout ; il est accompagné d'un exemple d'exécution dans un environnement C++ Builder.

```
#include <iostream.h>
main()
{
   int n = 25 ; long p = 250000; unsigned q = 63000 ;
   char c = 'a' ;
   float x = 12.3456789 ; double y = 12.3456789e16 ;
   char * ch = "bonjour" ;
   int * ad = & n ;

   cout << "valeur de n   : "  << n  << "\n" ;
   cout << "valeur de p   : "  << p  << "\n" ;
   cout << "caractère c   : "  << c  << "\n" ;
   cout << "valeur de q   : "  << q  << "\n" ;
   cout << "valeur de x   : "  << x  << "\n" ;
   cout << "valeur de y   : "  << y  << "\n" ;
   cout << "chaîne ch     : "  << ch << "\n" ;
   cout << "adresse de n  : " << ad << "\n" ;
   cout << "adresse de ch : " << (void *) ch << "\n" ;
}
```

```
valeur de n   : 25
valeur de p   : 250000
caractère c   : a
valeur de q   : 63000
valeur de x   : 12.345679
```

[4] - Les versions de C++ antérieures à la 2.0 traitaient un caractère comme un entier et, par suite, en affichaient le code.

[5] - Cette possibilité n'existait pas dans les versions de C++ antérieures à la 2.0.

```
valeur de y   : 1.234567e+17
chaîne ch     : bonjour
adresse de n  : 0x19c7fff4
adresse de ch : 0x19c700b4
```

Les possibilités d'écriture sur cout

Remarque :

Notez que C++ a décidé d'afficher la chaîne située à l'adresse indiquée par un pointeur de type char * et non l'adresse elle-même. Si tel n'avait pas été le cas, le comportement d'une instruction aussi banale que :

```
cout << "bonjour" ;
```

aurait été bien déroutant.

3. LECTURE SUR L'ENTRÉE STANDARD

3.1 Introduction

De même qu'il existe un flot de sortie prédéfini cout, associé à la sortie standard du C (stdout), il existe un flot d'entrée prédéfini, nommé cin, associé à l'entrée standard du C (stdin). De même que l'opérateur << permet d'envoyer des informations sur un flot de sortie (donc, en particulier, sur cout), l'opérateur >> permet de recevoir[6] de l'information en provenance d'un flot d'entrée (donc, en particulier, de cin).

Par exemple, l'instruction (n étant de type int) :

```
cin >> n ;
```

demandera de lire "des caractères" sur le flot cin et de les "convertir" en une valeur de type int.

D'une manière générale :

```
cin >> n >> p ;
```

sera équivalent à :

```
(cin >> n) >> p ;
```

Pour donner une interprétation imagée (et peu formelle) analogue à celle fournie pour cout, nous pouvons dire que la valeur de n est d'abord extraite du flot cin ; ensuite, la valeur de p est extraite du flot cin >> n (comme pour <<, le résultat de l'opérateur >> est un flot), c'est-à-dire de ce qu'est devenu le flot cin, après qu'on en a extrait la valeur de n.

6 - On dit aussi extraire

3.2 Les possibilités de lecture sur cin

D'une manière générale, dans toutes les versions de C++, vous pouvez utiliser l'opérateur >> pour accéder à des informations de type de base quelconque (signé ou non pour les types entiers) ou à des chaînes de caractères[7] (char *).

Par ailleurs, une bonne part des conventions d'analyse des caractères lus sont les mêmes que celles employées par scanf. Ainsi :

- les différentes informations sont séparées par un ou plusieurs caractères parmi ceux-ci[8] : espace, tabulation horizontale (\t) ou verticale (\v), fin de ligne (\n), retour chariot (\r) ou changement de page (\f)[9].
- un caractère "invalide" pour l'usage qu'on doit en faire (un point pour un entier, une lettre pour un nombre...) arrête l'exploration du flot, comme si l'on avait rencontré un séparateur ; mais ce caractère invalide sera à nouveau pris en compte lors d'une prochaine lecture.

En revanche, contrairement à ce qui se produisait pour scanf, la lecture d'un caractère sur cin commence par "sauter les séparateurs" ; aussi n'est-il pas possible de lire directement ces caractères. Nous verrons comment y parvenir dans le chapitre consacré aux flots.

3.3 Exemples

Nous vous proposons différents programmes, accompagnés d'exemples d'exécution (réalisés avec le clavier et l'écran comme entrées-sorties standard) illustrant les possibilités de lecture sur cin.

```
#include <iostream.h>
main()
{
   int n ; float x ;
   char t[81] ;
   do
     {  cout << "donnez un entier, une chaîne et un flottant : " ;
        cin >> n >> t >> x ;
        cout << "merci pour " << n << ", " << t << " et " << x << "\n" ;
     }
   while (n) ;
}
```

```
donnez un entier, une chaîne et un flottant : 15 bonjour 8.25
merci pour 15, bonjour et 8.25
donnez un entier, une chaîne et un flottant :       15

                          bonjour

                                  8.25
```

7. Il n'est pas prévu de lire la valeur d'un pointeur ; en pratique, cela n'aurait guère d'intérêt et, de plus, comporterait de grand risques.

8. - On les appelle parfois des "espaces-blancs" (de l'anglais, "white spaces").

9. - En pratique, on exploite essentiellement l'espace et la fin de ligne (dans les environnements DOS, toutefois, le retour chariot peut apparaître en plus de la fin de ligne, mais on ne s'en aperçoit que lorsqu'on lit des caractères).

```
merci pour 15, bonjour et 8.25
donnez un entier, une chaîne et un flottant : 0   bye   0
merci pour 0, bye et 0
```

Usage classique des séparateurs

```
#include <iostream.h>
main()
{
   char tc[81] ;           // pour conserver les caractères lus sur cin
   int i = 0 ;             // position courante dans le tableau c

   cout << "donnez une suite de caractères terminée par un point \n" ;
   do
      cin >> tc[i] ;       // attention, pas de test de débordement dans tc
   while (tc[i++] != '.') ;
   cout << "\n\nVoici les caractères effectivement lus : \n" ;

   i=0 ;
   do
      cout << tc[i] ;
   while (tc[i++] != '.') ;
}
```

```
donnez une suite de caractères terminée par un point
     Voyez comme

          C++
pose quelques problèmes
lors de la lecture d'une

     "suite de caractères"
    .

Voici les caractères effectivement lus :
VoyezcommeC++posequelquesproblèmeslorsdelalectured'une"suitedecaractères".
```

Quand on cherche à lire une suite de caractères

```
#include <iostream.h>
main()
{  int n ;
   do
```

```
      { cout << "donnez un nombre entier : " ;
        cin >> n ;
        cout << "voici son carré : " << n*n << "\n" ;
      }
   while (n) ;
}
```

```
donnez un nombre entier : 3
voici son carré : 9
donnez un nombre entier : à
voici son carré : 9
donnez un nombre entier : voici son carré : 9
donnez un nombre entier : voici son carré : 9
^C
```

Boucle infinie sur un caractère invalide

```
#include <iostream.h>
main()
{
   int n, p ;
   cout << "donnez une valeur pour n : " ;
   cin >> n ;
   cout << "merci pour " << n << "\n" ;
   cout << "donnez une valeur pour p : " ;
   cin >> p ;
   cout << "merci pour " << p << "\n" ;
}
```

```
donnez une valeur pour n : 12 25
merci pour 12
donnez une valeur pour p : merci pour 25
```

Quand le clavier et l'écran semblent mal synchronisés

IV. LES SPÉCIFICITÉS DU C++

Comme nous l'avons dit dans le premier chapitre, C++ dispose d'un certain nombre de spécificités qui ne sont pas véritablement axées sur la P.O.O. Il s'agit essentiellement des suivantes :

- nouvelle forme de commentaire (en fin de ligne),
- emplacement libre des déclarations,
- notion de référence,
- arguments par défaut dans les déclarations des fonctions,
- surdéfinition de fonctions,
- opérateurs new et delete,
- fonctions "en ligne" (inline).

D'une manière générale, ces possibilités seront, pour la plupart, souvent utilisées conjointement avec celles de P.O.O. C'est ce qui justifie que nous vous les exposions dès maintenant.

Notez que le C++ introduit également de nouveaux opérateurs de cast ; mais comme certains aspects interfèrent avec la notion d'objet et d'identification dynamique de type lors de l'exécution, ils feront l'objet d'une présentation séparée dans l'annexe C.

1. LE COMMENTAIRE DE FIN DE LIGNE

En C ANSI, un commentaire peut être introduit en n'importe quel endroit où un espace est autorisé[1] en le faisant précéder de /* et suivre de */. Il peut alors éventuellement s'étendre sur plusieurs lignes.

En C++, vous pouvez, *en outre*, utiliser des "commentaires de fin de ligne" en introduisant les deux caractères : //. Dans ce cas, tout ce qui est situé entre // et la fin de la ligne est un commentaire. Notez que cette nouvelle possibilité n'apporte qu'un surcroît de confort et de sécurité ; en effet, une ligne telle que :

```
cout << "bonjour\n" ; // formule de politesse
```

peut toujours être écrite ainsi :

---

1 - Donc, en pratique, n'importe où, pourvu qu'on ne "coupe pas en deux" un identificateur quelconque ou une chaîne constante.

```
cout << "bonjour\n" ; /*formule de politesse*/
```

Vous pouvez mêler (volontairement ou non !) les deux formules. Dans ce cas, notez que, dans :

```
/* partie1 // partie2 */ partie3
```

le commentaire "ouvert" par /* ne se termine qu'au prochain */ ; donc partie1 et partie2 sont des commentaires, tandis que partie3 est considérée comme appartenant aux instructions.

De même, dans :

```
partie1 // partie2 /* partie3 */ partie4
```

le commentaire introduit par // s'étend jusqu'à la fin de la ligne. Il "couvre" donc partie2, partie3 et partie4.

Remarques :

1) Le commentaire de fin de ligne constitue le seul cas où la fin de ligne joue un rôle significatif autre que celui d'un simple séparateur.

2) Si l'on utilise systématiquement le commentaire de fin de ligne, on peut alors faire appel à /* et * pour inhiber un ensemble d'instructions (contenant éventuellement des commentaires) en phase de mise au point.

2. DÉCLARATIONS ET INITIALISATIONS

2.1 Règles générales

C++ s'avère plus souple que le C ANSI en matière de déclarations. Plus précisément, en C++, il n'est plus obligatoire de regrouper au début les déclarations effectuées au sein d'une fonction ou au sein d'un bloc. Celles-ci peuvent être effectuées où bon vous semble, pour peu qu'elles apparaissent avant que l'on en ait besoin : leur portée reste limitée à la partie du bloc ou de la fonction suivant leur déclaration.

Par ailleurs, les expressions utilisées pour initialiser une variable scalaire peuvent être quelconques, alors qu'en C elles ne peuvent faire intervenir que des variables dont la valeur est connue dès l'entrée dans la fonction concernée.

Voici un exemple incorrect en C ANSI et accepté en C++ :

```
main()
{
   int n ;
     .....
   n = ...
     .....
   int q = 2*n - 1 ;
     .....
}
```

La déclaration "tardive" de q permet de l'initialiser[2] avec une expression dont la valeur n'était pas connue lors de l'entrée dans la fonction (ici main).

2.2 Cas des instructions structurées

L'instruction suivante est acceptée en C++ :

```
for (int i=0 ; ... ; ...)
       {
         .....
       }
```

Là encore, la variable i a été déclarée seulement au moment où l'on en avait besoin. Sa portée est, d'après la norme, limitée au bloc régit par l'instruction for. On notera qu'il n'existe aucune façon d'écrire des instructions équivalentes en C.

Cette possibilité s'applique à toutes les instructions structurées, c'est-à-dire aux instructions for, switch, while et do...while.

Remarque importante :

Le rôle de ces déclarations à l'intérieur d'instructions structurées n'a été fixé que tardivement par la norme ANSI. Dans les versions antérieures, ce genre de déclaration était, certes, autorisé mais, tout se passait comme si elle figurait à l'extérieur du bloc ; ainsi, l'exemple précédent était interprété comme si l'on avait écrit :

```
int i ;
 for (i=0 ; ... ; ... )
   {
     .....
   }
```

3. LA NOTION DE RÉFÉRENCE

En C, les arguments et la valeur de retour d'une fonction sont transmis par valeur. Pour simuler en quelque sorte ce qui se nomme "transmission par adresse" dans d'autres langages, il est alors nécessaire de "jongler" avec les pointeurs pour y parvenir (la transmission se faisant toujours par valeur, mais, dans ce cas, il s'agit de la valeur d'un pointeur). En C++, le principal intérêt de la notion de référence est qu'elle permet de laisser le compilateur mettre en œuvre les "bonnes instructions" permettant d'assurer effectivement un transfert par adresse. Pour mieux vous en faire saisir l'intérêt, nous vous proposons de faire d'abord un rappel montrant comment il fallait procéder en C.

3.1 Transmission des arguments en C

Considérons l'exemple classique suivant :

---

2 - N'oubliez pas qu'en C (comme en C++) il est possible d'initialiser une variable automatique scalaire à l'aide d'une expression quelconque.

```
#include <iostream.h>
main()
{
   void echange (int, int) ;
   int n=10, p=20 ;
   cout << "avant appel : " << n << " " << p << "\n" ;
   echange (n, p) ;
   cout << "après appel : " << n << " " << p << "\n" ;
}
void echange (int a, int b)
{
   int c ;
   cout << "début echange : " << a << " " << b << "\n" ;
   c = a ; a = b ; b = c ;
   cout << "fin echange   : " << a << " " << b << "\n" ;
}
```

```
avant appel : 10 20
début echange : 10 20
fin echange   : 20 10
après appel : 10 20
```

Les arguments sont transmis par valeur

Lors de l'appel de echange, il y a transmission des valeurs de n et de p ; on peut considérer que la fonction les a recopiées dans des emplacements locaux, correspondant à ses arguments formels a et b et qu'elle a effectivement "travaillé" sur ces copies.

Bien entendu, il est toujours possible de programmer une fonction echange pour qu'elle opère effectivement sur des variables de la fonction qui l'appelle ; il suffit tout simplement de lui fournir, en argument, l'adresse de ces variables, comme dans l'exemple suivant :

```
#include <iostream.h>
main()
{
   void echange (int *, int *) ;
   int n=10, p=20 ;
   cout << "avant appel : " << n << " " << p << "\n" ;
   echange (&n, &p) ;
   cout << "après appel : " << n << " " << p << "\n" ;
}
void echange (int * a, int * b)
{
   int c ;
   cout << "début echange : " << * a << " " << * b << "\n" ;
   c = * a ; * a = * b ; * b = c ;
   cout << "fin echange   : " << * a << " " << * b << "\n" ;
```

```
}
```

```
avant appel : 10 20
début echange : 10 20
fin echange   : 20 10
après appel : 20 10
```

Mise en œuvre, par le programmeur, d'une transmission par adresse

Notez bien les différences entre les deux exemples, à la fois dans l'écriture de la fonction echange, mais aussi dans son appel. Ce dernier point signifie que l'utilisateur de la fonction (qui n'est pas toujours celui qui l'a écrite) doit savoir s'il faut lui transmettre une variable ou son adresse[3].

3.2 Transmission par référence en C++

C++ permet de demander au compilateur de prendre lui-même en charge la transmission des arguments par adresse : on parle alors en général, dans ce cas, de transmission d'argument par référence. Le même mécanisme pourra également s'appliquer à une valeur de retour d'une fonction. Examinons ces deux possibilités.

a) Transmission d'arguments par référence

Le programme ci-dessous montre comment appliquer un tel mécanisme à notre fonction echange.

```
#include <iostream.h>
main()
{  void echange (int &, int &) ;
   int n=10, p=20 ;
   cout << "avant appel : " << n << " " << p << "\n" ;
   echange (n, p) ;        // attention, ici pas de &n, &p
   cout << "après appel : " << n << " " << p << "\n" ;
}
void echange (int & a, int & b)
{  int c ;
   cout << "début echange : " << a << " " << b << "\n" ;
   c = a ; a = b ; b = c ;
   cout << "fin echange   : " << a << " " << b << "\n" ;
}
```

```
avant appel : 10 20
début echange : 10 20
fin echange   : 20 10
après appel : 20 10
```

3 - Certes, ici, si l'on veut que la fonction echange puisse "faire correctement son travail", le choix s'impose. Mais, il n'en va pas toujours ainsi.

Dans l'instruction :

```
void echange (int & a, int & b) ;
```

La notation int & a signifie que a est une information entière transmise par référence. Notez bien que, dans la fonction echange, on utilise simplement le symbole a pour désigner cette variable dont la fonction aura reçu effectivement l'adresse (et non la valeur) : il n'est plus utile (et ce serait une erreur !) de faire appel à l'opérateur d'indirection *.

Autrement dit, il **nous suffit d'avoir fait ce choix de transmission par référence au niveau de l'en-tête de la fonction pour que le processus soit entièrement pris en charge par le compilateur**[4].

Le même phénomène s'applique au niveau de l'utilisation de la fonction. Il suffit, en effet, d'avoir spécifié, dans le prototype, les arguments (ici, les deux) que l'on souhaite voir transmis par référence. Au niveau de l'appel :

```
echange (n, p) ;
```

nous n'avons plus à nous préoccuper du mode de transmission effectivement utilisé.

Remarques :

1) La transmission d'arguments par référence simplifie l'écriture de la fonction correspondante.
2) Le choix du mode de transmission par référence est fait au moment de l'écriture de la fonction concernée. L'utilisateur de la fonction n'a plus à s'en soucier ensuite, si ce n'est au niveau de la déclaration du prototype de la fonction (en général, d'ailleurs, ce prototype viendra d'un fichier en-tête).
3) En contrepartie des avantages précédents, l'emploi de la transmission par référence accroît les risques "d'effets de bord" non désirés. En effet, lorsqu'il appelle une fonction, l'utilisateur ne sait plus s'il transmet, au bout du compte, la valeur ou l'adresse d'un argument (la même notation pouvant désigner l'une ou l'autre des deux possibilités) ; il risque donc de modifier une variable dont il pensait n'avoir transmis qu'une copie de la valeur !

Remarque :

Lorsqu'on doit transmettre en argument la référence à un pointeur, on est amené à utiliser ce genre d'écriture :

```
int * & adr       // adr est une référence à un pointeur sur un int
```

b) Transmission par référence de la valeur de retour d'une fonction

Le mécanisme que nous venons d'exposer pour la transmission des arguments s'applique également à la valeur de retour d'une fonction. Son véritable intérêt n'apparaîtra toutefois que lorsque nous étudierons la

4 - Cette possibilité est analogue à l'utilisation du mot clé var dans l'en-tête d'une procédure en Pascal.

surdéfinition d'opérateurs ; en effet, dans certains cas, il sera indispensable qu'un opérateur (c'est-à-dire, en fait, une fonction) transmette sa valeur de retour par référence[5].

3.3 La référence d'une manière générale

L'essentiel concernant la notion de référence a été présenté dans le paragraphe précédent. Cependant, en toute rigueur, la notion de référence peut intervenir en dehors de la notion d'argument ou de valeur de retour. C'est ce que nous vous présentons dans le présent paragraphe (qui peut être ignoré sans problèmes dans une première lecture).

a) La notion de référence est plus générale que celle d'argument

D'une manière générale, il est possible de déclarer un identificateur comme référence d'une autre variable. Considérez, par exemple, ces instructions :

```
int n ;
int & p = n ;
```

La seconde signifie que p est une référence à la variable n. Ainsi, dans la suite, n et p désigneront le même emplacement mémoire. Par exemple, avec :

```
n = 3 ;
cout << p ;
```

nous obtiendrons la valeur 3.

b) Initialisation de référence

La déclaration :

```
int & p=n ;
```

est en fait une déclaration de référence (ici p) accompagnée d'une initialisation (à la référence de n). D'une façon générale, il n'est pas possible de déclarer une référence sans l'initialiser, comme dans :

```
int & p ;     // incorrect, car pas d'initialisation
```

Notez bien qu'une fois déclarée (et initialisée), une référence ne peut plus être modifiée. D'ailleurs aucun mécanisme n'est prévu à cet effet : si, ayant déclaré int & p= n ; vous écrivez p= q, il s'agit obligatoirement de l'affectation de la valeur de q à l'emplacement de référence p, et non de la modification de la référence q.

Une déclaration telle que :

```
int & n = 3 ;   // incorrecte depuis la version 3
```

est incorrecte depuis la version 3. Ceci se justifie par le fait que si elle était acceptée, elle reviendrait à initialiser n avec une référence à la valeur 3. Dans ces conditions, que ferait l'instruction :

---

5 - Ce sera notamment le cas de l'opérateur [ ] lorsque l'on souhaitera pouvoir l'employer dans des affectations de la forme t[i] = x.

```
n = 5 ;
```

En revanche, une déclaration telle que celle-ci est correcte :

```
const int & n = 3 ;
```

Elle génère une variable temporaire (ayant une durée de vie imposée par l'emplacement de la déclaration) contenant la valeur 3 et place sa référence dans n. On peut dire que tout se passe comme si vous aviez :

```
int temps = 3 ;
int & n = temps ;
```

avec cette différence que, dans le premier cas, vous n'avez pas explicitement accès à la variable temporaire.

c) La transmission d'un argument ou d'une valeur de retour est une initialisation

Cette remarque prend tout son sens dans le cas où l'on a affaire à des références. En effet, supposons qu'une fonction fct ait pour prototype :

```
void fct (int &) ;
```

Alors, le compilateur refusera un appel de la forme suivante (n étant de type int) :

```
fct (3) ;      // incorrect
```

Il en ira de même pour :

```
const int c = 15 ;
  ...
fct (c) ;      // incorrect
```

En effet, de tels appels sont assimilés à des initialisations de références (à quelque chose de non constant) avec une référence à une constante. Ce refus se justifie parfaitement si l'on pense que l'on doit transmettre à fct l'adresse d'une constante (3 ou c), sachant que cette fonction risque de modifier ce qui se trouve à cette adresse.

Pour les mêmes raisons, l'appel suivant sera rejeté :

```
fct (&n) ;
```

En revanche, avec une fonction de prototype :

```
fct1 (const int &) ;
```

ces appels seront corrects :

```
fct1 (3) ;     // correct ici
fct1 (c) ;     // correct ici
```

Qui plus est, dans ce cas, un appel tel fct1 (x) sera accepté quel que soit le type de x. En effet, dans ce cas, il y a création d'une variable temporaire (de type int) qui recevra le résultat de la conversion de x en int.

Cette fois, l'acceptation de ces instructions se justifie par le fait que fct a prévu de recevoir une référence à quelque chose de constant ; le risque de modification évoqué précédemment n'existe donc plus.

4. LES ARGUMENTS PAR DÉFAUT

4.1 Exemples

En C ANSI, il est indispensable que l'appel d'une fonction contienne autant d'arguments que la fonction en attend effectivement. C++ permet de s'affranchir en partie de cette règle, grâce à un mécanisme d'attribution de valeurs par défaut à des arguments. Considérez l'exemple suivant :

```
#include <iostream.h>
main()
{
   int n=10, p=20 ;
   void fct (int, int=12) ; // proto avec une valeur par défaut
   fct (n, p) ;             // appel "normal"
   fct (n) ;                // appel avec un seul argument
                            // fct()  serait, ici, rejeté */
}
void fct (int a, int b)     // en-tête "habituelle"
{
   cout << "premier argument : " << a << "\n" ;
   cout << "second argument  : " << b << "\n" ;
}
```

```
premier argument : 10
second argument  : 20
premier argument : 10
second argument  : 12
```

Exemple de définition de valeur par défaut pour un argument

La déclaration de fct, ici dans la fonction main, est réalisée par le prototype :

```
void fct (int, int = 12) ;
```

Vous y notez la déclaration du second argument sous la forme :

```
int = 12
```

Celle-ci précise au compilateur que, en cas d'absence de ce second argument dans un éventuel appel de fct, il lui faudra "faire comme si" l'appel avait été effectué avec cette valeur.

Les deux appels de fct illustrent le phénomène. Notez qu'un appel tel que :

```
fct ( )
```

serait rejeté à la compilation puisqu'ici il n'était pas prévu de valeur par défaut pour le premier argument de fct.

Voici un second exemple, dans lequel nous avons prévu des valeurs par défaut pour tous les arguments de fct.

```
#include <iostream.h>
main()
{
   int n=10, p=20 ;
   void fct (int=0, int=12) ; // proto avec deux valeurs par défaut
   fct (n, p) ;               // appel "normal"
   fct (n) ;                  // appel avec un seul argument
   fct () ;                   // appel sans argument
}

void fct (int a, int b)     // en-tête "habituelle"
{
   cout << "premier argument : " << a << "\n" ;
   cout << "second argument  : " << b << "\n" ;
}
```

```
premier argument : 10
second argument  : 20
premier argument : 10
second argument  : 12
premier argument : 0
second argument  : 12
```

Exemple de définition de valeurs par défaut pour plusieurs arguments

4.2 D'une manière générale

Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste. Par exemple, une déclaration telle que :

```
float fexple (int = 5, long, int = 3) ;
```

est interdite. En fait, une telle interdiction relève du pur bon sens : en effet, si cette déclaration était acceptée, l'appel suivant :

```
fexple (10, 20) ;
```

pourrait être interprété aussi bien comme :

```
fexple (5, 10, 20) ;
```

que comme :

```
fexple (10, 20, 3) ;
```

Notez bien que le mécanisme proposé par C++ revient à **fixer les valeurs par défaut dans la déclaration de la fonction et non dans sa définition.** Autrement dit, ce n'est pas le "concepteur" de la fonction qui

décide des valeurs par défaut, mais l'utilisateur. Une conséquence immédiate de cette particularité est que les arguments soumis à ce mécanisme et les valeurs correspondantes peuvent varier d'une utilisation à une autre ; en pratique, toutefois, ce point ne sera guère exploité, ne serait-ce que parce que les déclarations de fonctions sont, en général, "figées" une fois pour toutes, dans un fichier en-tête.

Nous verrons que les arguments par défaut se révéleront particulièrement précieux lorsqu'il s'agira de fabriquer ce que l'on nomme le "constructeur d'une classe".

Remarques :

1) Théoriquement C++ autorise (au sein d'un même fichier source) une redéclaration de la fonction, dans la mesure où cette dernière se contente d'ajouter de nouvelles valeurs par défaut. Pour notre part, nous préférons nous en tenir aux points proposés, pour d'évidentes raisons de simplicité.

2) Si l'on souhaite attribuer une valeur par défaut à un argument de type pointeur, on prendra garde de séparer par au moins un espace les caractères * et = ; dans le cas contraire, ils seraient interprétés comme l'opérateur d'affectation *=, ce qui conduirait à une erreur.

3) Les valeurs par défaut peuvent être constituées de n'importe quelle expression (et non seulement d'une expression constante), pour peu que cette dernière ait un sens au moment où elle est compilée. En voici un exemple d'école (il nécessite simplement que les déclarations de x et de n figurent avant celle de fct) :

```
float x ;
int n ;
 void fct (float = n*x + 1.5) ;     /* la valeur par défaut dépend des
variables
                                          n et x */
```

4.3 Lorsque l'on conjugue argument par défaut et transmission par référence

N.B. Ce paragraphe peut être ignoré dans un premier temps. Sachez simplement que sa lecture nécessite l'étude préalable du paragraphe 3.3.

Voici, en parallèle, deux exemples de programmes qui réalisent la même chose, l'un en faisant appel à la transmission d'un argument par référence, l'autre à une démarche classique basée sur un pointeur.

```
main()
{
   int n, p ;
   void fct (int, int & = n) ;
   fct(10, p) ;      // idem p=10
   fct(10) ;         // idem n=10
}
void fct (int a, int & b)
{
   b = a ;
}
```

```
main()
{
   int n, p ;
   void fct (int, int * = &n) ;
   fct(10, &p) ;      // idem p=10
   fct(10) ;          // idem n=10
}
void fct (int a, int * adb)
{
   * adb = a ;
}
```

Exemple de valeur par défaut pour un argument transmis par référence ou par adresse

Notez la déclaration :

```
void fct (int, int & = n) ;
```

dans laquelle la notation

```
int & = n
```

signifie :

- que l'argument correspondant, de type int, est transmis par référence,
- que cette référence reçoit une valeur par défaut correspondant à l'adresse de n (et non à sa valeur).

Si nous avions souhaité que cet argument (transmis par référence) reçoive par défaut la valeur 3, nous n'aurions pas pu déclarer ainsi notre fonction fct :

```
void fct (int, int & = 3) ;      // erreur
```

En revanche, cela aurait été possible si le second argument de notre fonction avait reçu l'attribut const (et, bien sûr, l'en-tête approprié) :

```
void fct (int, const & = 3) ;    // correct
```

Ceci aurait, comme nous l'avons déjà indiqué dans le paragraphe précédent, entraîné la création d'une variable temporaire, contenant la valeur 3, dont on aurait transmis la référence comme argument par défaut.

5. SURDÉFINITION DE FONCTIONS

D'une manière générale, on parle de "surdéfinition"[1] lorsqu'un même symbole possède plusieurs significations différentes, le choix de l'une des significations se faisant en fonction du "contexte". C'est ainsi que C, comme la plupart des langages évolués, utilise la surdéfinition d'un certain nombre d'opérateurs. Par exemple, dans une expression telle que :

```
a + b
```

la signification du + dépend du type des opérandes a et b ; suivant le cas, il pourra s'agir d'une addition d'entiers ou d'une addition de flottants. De même, le symbole * peut désigner, suivant le contexte, une multiplication d'entiers, de flottants ou une "indirection[2]".

Un des grands atouts de C++ est de permettre la surdéfinition de la plupart des opérateurs (lorsqu'ils sont associés à la notion de classe). Lorsque nous étudierons cet aspect, nous verrons alors qu'il repose en fait sur la surdéfinition de fonctions. C'est cette dernière possibilité que nous proposons d'étudier ici pour elle-même.

Pour pouvoir employer plusieurs fonctions de même nom, il faut, bien sûr, un critère (autre que le nom) permettant de choisir la bonne fonction. En C++, ce choix est basé (comme pour les opérateurs cités

1 - De "overloading", en anglais, parfois traduit par "surcharge".

2 - On parle parfois de "déréférence" ; cela correspond à des situations telles que :

```
int* a ;
*a = 5;
```

précédemment en exemple) sur le type des arguments. Nous commencerons par vous présenter un exemple complet montrant comment mettre en œuvre la surdéfinition de fonctions. Nous examinerons ensuite différentes situations d'appel d'une fonction surdéfinie avant d'étudier les règles détaillées qui président au choix de la "bonne fonction".

5.1 Mise en œuvre de la surdéfinition de fonctions

Nous allons définir et utiliser deux fonctions nommées, toutes les deux, sosie. La première possédera un argument de type int, la seconde un argument de type double, ce qui les différencie bien l'une de l'autre. Pour que l'exécution du programme montre clairement la fonction effectivement appelée, nous introduisons dans chacune des fonctions une instruction d'affichage appropriée. Dans le programme "d'essai", nous nous contentons d'appeler successivement la fonction surdéfinie sosie, une première fois avec un argument de type int, une seconde fois avec un argument de type double.

```
#include <iostream.h>

void sosie (int) ;            // les prototypes
void sosie (double) ;

main()                        // le programme de test
{
   int n=5, p ; double x=2.5 ;
   sosie (n) ;
   sosie (x) ;
}

void sosie (int a)            // la première fonction
{
   cout << "sosie numéro I   a = " << a << "\n" ;
}
void sosie (double a)         // la deuxième fonction
{
   cout << "sosie numéro II  a = " << a << "\n" ;
}
```

```
sosie numéro I   a = 5
sosie numéro II  a = 2.5
```

Exemple de surdéfinition de la fonction sosie

Vous constatez que le compilateur a bien mis en place l'appel de la "bonne fonction" sosie, au vu de la liste d'arguments (ici réduite à un seul).

5.2 Exemples de choix d'une fonction surdéfinie

Notre précédent exemple était simple, dans la mesure où nous appelions toujours la fonction sosie avec un argument ayant **exactement** l'un des types prévus dans les prototypes (int ou double). On peut se demander ce qui se produirait si nous l'appelions, par exemple, avec un argument de type char, long ou pointeur, ou si l'on avait affaire à des fonctions comportant plusieurs arguments...

Avant d'examiner les règles de détermination d'une fonction surdéfinie, examinons tout d'abord quelques situations assez intuitives.

Exemple 1

Avec nos deux précédentes déclarations :

```
void sosie (int) ;        // sosie I
void sosie (double) ;     // sosie II
```

et les variables suivantes :

```
char c ; float y ;
```

- l'instruction sosie(c) appellera la fonction sosie I, après conversion de la valeur de c en int,
- l'instruction sosie(y) appellera la fonction sosie II, après conversion de la valeur de y en double,
- l'instruction sosie('d') appellera la fonction sosie I, après conversion de la valeur de 'd' en int.

Exemple 2

Avec ces déclarations :

```
void affiche (char *) ;     // affiche I
void affiche (void *) ;     // affiche II
char * ad1 ;
double * ad2 ;
```

- l'instruction affiche (ad1) appellera la fonction affiche I,
- l'instruction affiche (ad2) appellera la fonction affiche II, après conversion de la valeur de ad2 en void*.

Exemple 3

Avec ces déclarations :

```
void essai (int, double) ;     // essai I
void essai (double, int) ;     // essai II
int n, p ; double z ; char c ;
```

- l'instruction essai(n,z) appellera la fonction essai I,
- l'instruction essai(c,z) appellera la fonction essai I, après conversion de la valeur de c en int,
- l'instruction essai(n,p) conduira à une erreur de compilation, compte tenu de son ambiguïté ; en effet, deux possibilités existent ici : convertir p en double, sans modifier n et appeler essai I ou, au contraire, convertir n en double, sans modifier p et appeler essai II.

Exemple 4

Avec ces déclarations :

```
void test (int n=0, double x=0) ;      // test I
void test (double y=0, int p=0) ;      // test II
int n ; double z ;
```

- l'instruction test(n,z) appellera la fonction test I
- l'instruction test(z,n) appellera la fonction test II
- l'instruction test(n) appellera la fonction test I
- l'instruction test(z) appellera la fonction test II
- l'instruction test() conduira à une erreur de compilation, compte tenu de son ambiguïté.

Exemple 5

Avec ces déclarations :

```
void truc (int) ;          // truc I
void truc (const int) ;    // truc II
```

vous obtiendrez une erreur de compilation. En effet, C++ n'a pas prévu de distinguer int de const int. Ceci se justifie par le fait que, les deux fonctions truc recevant une copie de l'information à traiter, aucun risque n'existe de modifier la valeur originale.

Exemple 6

En revanche, considérez maintenant ces déclarations :

```
void chose (int *) ;        // chose I
void chose (const int *) ; // chose II
int n = 3 ;
const p = 5 ;
```

Cette fois, la distinction entre int * et const int * est justifiée. En effet, on peut très bien prévoir que chose I modifie la valeur de la lvalue[3] dont elle reçoit l'adresse, tandis que chose II n'en fait rien. Cette distinction est possible en C+, de sorte que :

- l'instruction chose (&n) appellera la fonction chose I
- l'instruction chose (&p) appellera la fonction chose II

[3] - Rappelons qu'on nomme lvalue la référence à quelque chose dont on peut modifier la valeur. Ce terme provient de la contraction de "left value" qui désigne quelque chose qui peut apparaître à gauche d'un opérateur d'affectation.

5.3 Règles de recherche d'une fonction surdéfinie

Pour l'instant, nous vous proposons simplement de vous en donner plutôt la philosophie générale, ce qui sera suffisant pour l'étude des chapitres suivants. Au fil de l'étude de cet ouvrage, nous serons amené à vous apporter des informations complémentaires. De plus, l'annexe A reprend l'ensemble précis de toutes ces règles.

a) Cas des fonctions à un argument

Le compilateur recherche la "meilleure correspondance" possible. Bien entendu, pour pouvoir définir ce qu'est cette meilleure correspondance, il faut qu'il dispose d'un critère d'évaluation. Pour ce faire, il est prévu différents niveaux de correspondance :

1 - Correspondance exacte : on distingue bien les uns des autres les différents types de base, en tenant compte de leur éventuel attribut de signe[4] ; de plus, comme on l'a vu dans nos précédents exemples, l'attribut const peut intervenir dans le cas de pointeurs ou de références.

2 - Corrrespondance avec "promotions numériques", c'est-à-dire essentiellement :

   * char et short -> int

   * float -> double

3 - Conversions dites "standard" : il s'agit des conversions légales en C++, c'est-à-dire de celles qui peuvent être imposées par une affectation (sans opérateur de "cast") ; cettte fois, il peut s'agir de conversions dégradantes puisque, notamment, toute conversion d'un type numérique en un autre type numérique est acceptée.

   D'autres niveaux sont prévus ; en particulier on pourra faire intervenir ce que l'on nomme des "conversions définies par l'utilisateur" (C.D.U.) ; elles ne seront étudiées que dans le chapitre X.

La recherche s'arrête au premier niveau ayant permis de trouver une correspondance et cette dernière doit être unique ; si plusieurs fonctions conviennent au même niveau de correspondance, il y a erreur de compilation due à l'ambiguïté rencontrée. Bien entendu, si aucune fonction ne convient à aucun niveau, il y a également erreur de compilation.

b) Cas des fonctions à plusieurs arguments

L'idée générale est qu'il doit se dégager une fonction "meilleure" que toutes les autres. Pour ce faire, le compilateur sélectionne, **pour chaque argument**, la ou les fonctions qui réalisent la meilleure correspondance (au sens de la hiérarchie définie ci-dessus). Parmi l'ensemble des fonctions ainsi sélectionnées, il choisit celle (si elle existe et si elle est unique) qui réalise, pour chaque argument, une correspondance au moins égale à celle de toutes les autres fonctions[5].

Si plusieurs fonctions conviennent, là encore, on aura une erreur de compilation due à l'ambiguïté rencontrée ; de même si aucune fonction ne convient, il y aura erreur de compilation.

Notez que les fonctions comportant un ou plusieurs arguments par défaut sont traitées comme si plusieurs fonctions différentes avaient été définies avec un nombre croissant d'arguments.

---

4 - Attention, en C++, char est différent de signed char et de unsigned char.

5 - Ce qui revient à dire qu'il considère l'intersection des ensembles constitués des fonctions réalisant la meilleure correspondance pour chacun des arguments.

5.4 Le mécanisme de la surdéfinition de fonctions

Jusqu'ici, nous avons examiné la manière dont le compilateur faisait le choix de la "bonne fonction", en raisonnant sur un seul fichier source à la fois. Mais il serait tout à fait envisageable :

- de compiler d'une part un fichier contenant la définition de nos deux fonctions nommées sosie,
- d'utiliser ces fonctions par ailleurs en nous contentant d'en fournir les prototypes.

Or, pour que ceci soit possible, l'éditeur de liens doit être en mesure d'effectuer le lien entre le choix opéré par le compilateur et la "bonne fonction" figurant dans un autre module objet. Cette reconnaissance est basée sur la modification, par le compilateur, des noms "externes" des fonctions ; celui-ci fabrique un nouveau nom basé, de manière déterministe, d'une part sur le nom interne de la fonction, d'autre part sur le nombre et la nature de ses arguments.

Il est très important de noter que ce mécanisme s'applique à toutes les fonctions, qu'elles soient effectivement surdéfinies ou non (il est impossible de savoir si une fonction compilée dans un fichier source sera surdéfinie dans un autre). On voit donc qu'un problème se pose, dès lors que l'on souhaite pouvoir utiliser, dans un programme C++, une fonction écrite et compilée en C (ou dans un autre langage utilisant les mêmes conventions d'appels de fonction, notamment l'assembleur ou le Fortran) ; en effet, une telle fonction ne voit pas son nom modifié suivant le mécanisme évoqué. Une solution existe toutefois : déclarer une telle fonction, en faisant précéder son prototype de la mention extern "C". Par exemple, si nous avons écrit et compilé en C une fonction d'en-tête :

```
double fct (int n, char c) ;
```

et que nous souhaitons l'utiliser dans un programme C++, il nous suffira d'y fournir son prototype de la façon suivante :

```
extern "C" double fct (int, char) ;
```

Remarques :

1) Il existe une forme "collective" de la déclaration extern ; elle se présente ainsi :

```
extern "C" { void exple (int) ;
             double chose (int, char, float) ;
             .....
           } ;
```

2) Le problème évoqué pour les fonctions C (assembleur ou Fortran) se pose, a priori, pour toutes les fonctions de la bibliothèque standard C que l'on réutilise en C++ ; en fait, dans beaucoup d'environnements, cet aspect est automatiquement pris en charge au niveau des fichiers en-tête correspondants (ils contiennent des déclarations extern conditionnelles).

3) Il est possible d'employer, au sein d'un même programme C++, une fonction C (assembleur ou Fortran) et une ou plusieurs autres fonctions C++ de même nom (mais d'arguments différents). Par exemple, nous pouvons utiliser dans un programme C++ la fonction fct précédente et deux fonctions C++ d'en-tête :

```
void fct (double x)
void fct (float y)
```

en procédant ainsi :

```
extern "C" void fct (int) ;
void fct (double) ;
void fct (float) ;
```

Suivant la nature de l'argument d'appel de fct, il y aura bien appel de l'une des trois fonctions fct. Notez toutefois qu'il n'est pas possible de mentionner plusieurs fonctions C de nom fct.

6. LES OPÉRATEURS NEW ET DELETE

En langage C, la gestion dynamique de mémoire fait appel à des fonctions de la bibliothèque standard telles que malloc et free. Bien entendu, celles-ci, comme toutes les fonctions standard, restent utilisables en C++.

Mais, dans le contexte de la Programmation Orientée Objet, C++ a introduit deux nouveaux opérateurs, new et delete, particulièrement adaptés à la gestion dynamique d'objets. Ces opérateurs peuvent également être utilisés pour des "variables classiques[6]". Dans ces conditions, par souci d'homogénéité et de simplicité, il est plus raisonnable, en C++, d'utiliser systématiquement ces opérateurs, que l'on ait affaire à des variables classiques ou à des objets. C'est pourquoi nous vous les présentons dès maintenant.

6.1 Exemples d'utilisation de new

a) Avec la déclaration :

```
int * ad ;
```

l'instruction :

```
ad = new int ;
```

permet d'allouer l'espace mémoire nécessaire pour un élément de type int et d'affecter à ad l'adresse correspondante. En C, vous auriez obtenu le même résultat en écrivant :

```
ad = (int *) malloc (sizeof (int)) ;
```

(l'opérateur de "cast", ici int*, étant facultatif)

Compte tenu de ce qu'en C++, les déclarations ont un emplacement libre, vous pouvez même déclarer la variable ad au moment où vous en avez besoin en écrivant, par exemple :

```
int * ad = new int ;
```

b) Avec la déclaration :

```
char * adc ;
```

l'instruction :

```
adc = new char[100] ;
```

alloue l'emplacement nécessaire pour un tableau de 100 caractères et place l'adresse (de début) dans adc. En C, vous auriez obtenu le même résultat en écrivant

```
adc = (char *) malloc (100) ;
```

6 - Un objet étant en fait une variable d'un type particulier.

6.2 Syntaxe et rôle de new

new est donc un opérateur unaire (à un seul opérande) qui s'utilise ainsi :

```
new type
```

où type représente un type absolument quelconque. Il fournit comme résultat :

- un pointeur (de type type*) sur l'emplacement correspondant, lorsque l'allocation a réussi,
- un pointeur nul (NULL = 0), dans le cas contraire.

new accepte également une syntaxe de la forme :

```
new type[n]
```

où n désigne une expression entière quelconque (non négative).

Dans ce cas, new alloue l'emplacement nécessaire pour n éléments du type indiqué et fournit en résultat un pointeur (toujours de type type *) sur le premier élément de ce tableau.

Remarques :

1) La norme de C++ prévoit qu'en cas d'échec, new peut[7] déclencher ce que l'on nomme une exception de type bad_alloc. Ce mécanisme de gestion des exceptions est éludié en détail dans le chapitre XVII. Vous y verrez que si rien n'est prévu par le programmeur pour traiter une exception, le programme s'interrompt. Autrement dit, on ne peut plus compter (de façon certaine) sur un pointeur nul dans les implémentations respectant la norme à venir, à moins de prendre soi-même en compte la gestion de l'exception bad_alloc. Si vous ne souhaitez pas faire cette démarche (au demeurant relativement lourde !), sachez que le paragraphe 6.5 vous proposera une solution plus simple.

2) En toute rigueur, new peut être utilisé pour allouer un emplacement pour un tableau à plusieurs dimensions, par exemple :

   ```
   new type[n][10]
   ```

   Dans ce cas, new fournit un pointeur de type type * [10]. D'une manière générale, la première dimension peut être une expression entière quelconque ; les autres doivent obligatoirement être des expressions constantes.

   Cette possibilité est rarement utilisée en pratique.

6.3 Exemples d'utilisation de l'opérateur delete

Lorsque l'on souhaite libérer un emplacement alloué préalablement par new, on doit absolument utiliser l'opérateur delete. Ainsi, pour libérer les emplacements créés dans les exemples du paragraphe 6.1, nous écrivons :

```
delete ad ;
```

pour l'emplacement alloué par :

```
ad = new int ;
```

et :

7 - Actuellement, la norme ne l'impose pas.

```
delete adc ;
```

pour l'emplacement alloué par :

```
adc = new char [100] ;
```

6.4 Syntaxe et rôle de l'opérateur delete

La syntaxe usuelle de l'opérateur delete est la suivante (adresse étant une expression devant avoir comme valeur un pointeur sur un emplacement alloué par new) :

delete adresse

Notez bien que le comportement du programme n'est absolument pas défini lorsque :

- vous libérez, par new, un emplacement déjà libéré ; nous verrons, à ce propos, que certaines précautions devront être prises lorsque l'on définit des constructeurs et des destructeurs de certains objets,
- vous fournissez à new, une "mauvaise adresse" où un pointeur obtenu autrement que par new (malloc, par exemple).

Remarque :

Il existe une autre syntaxe de new ; de la forme delete [] adresse, elle n'intervient que dans le cas de tableaux d'objets. Nous en parlerons dans le paragraphe 6 du chapitre 7.

6.5 Pour gérer les débordements de mémoire : set_new_handler

Dans le paragraphe 6.2, nous avons vu qu'il n'était pas toujours possible de détecter le manque d'espace mémoire en examinant la valeur de retour de new.

En fait, C++ vous permet de définir une fonction de votre choix et de demander qu'elle soit appelée en cas de manque de mémoire. Il vous suffit, pour ce faire, d'appeler la fonction set_new_handler, en lui fournissant, en argument, l'adresse de la fonction que vous avez prévue pour traiter le cas de manque de mémoire. Voici, par exemple, un programme qui alloue des emplacements pour des tableaux d'entiers dont la taille est fournie en donnée, et ceci jusqu'à ce qu'il n'y ait plus suffisamment de place (notez qu'ici nous utilisons toujours la même variable adr pour recevoir les différentes adresses des tableaux, ce qui, dans un programme réel, ne serait probablement pas acceptable).

```
#include <iostream.h>
main()
{
   void deborde () ;   // proto fonction appelée en cas manque mémoire
   set_new_handler (&deborde) ;
   long taille ;
   int * adr ;
   int nbloc ;
   cout << "Taille souhaitée ? " ;
   cin >> taille ;
   for (nbloc=1 ; ; nbloc++)
```

```
        { adr = new int [taille] ;
          cout << "Allocation bloc numéro : " << nbloc << "\n" ;
        }
}
void deborde ()       // fonction appelée en cas de manque mémoire
{  cout << "Mémoire insuffisante - arrêt exécution \n " ;
   exit (1) ;
}
```

```
Taille souhaitée ? 5000
Allocation bloc numéro : 1
Allocation bloc numéro : 2
Allocation bloc numéro : 3
Allocation bloc numéro : 4
Allocation bloc numéro : 5
Mémoire insuffisante - arrêt exécution
```

Exemple de définition, par set_new_handler, d'une fonction de gestion du manque de mémoire

Remarque :

La technique décrite ici est utilisable avec toutes les versions de C++, qu'elles disposent ou non du mécanisme de gestion des exceptions. En effet, dans ce dernier cas, l'exception de type bad_alloc est lancée, non pas directement par new, mais par l'intermédiaire d'une fonction.

7. LA SPÉCIFICATION INLINE

7.1 Rappels concernant les macros et les fonctions

En langage C, vous savez qu'il existe deux notions assez voisines, à savoir les macros et les fonctions. Une macro et une fonction s'utilisent apparemment de la même façon en faisant suivre leur nom d'une liste d'arguments entre parenthèses. Cependant :

- les instructions correspondant à une macro sont incorporées à votre programme[8], à chaque fois que vous l'appelez ;

- les instructions correspondant à une fonction sont "générées" une seule fois[9] ; à chaque appel, il y aura seulement mise en place des instructions nécessaires à établir la liaison entre le programme[10] et la fonction, c'est-à-dire : sauvegarde de "l'état courant" (valeurs de certains registres, par exemple), recopie des valeurs des arguments, branchement avec conservation de l'adresse de retour..., recopie de la valeur de retour, restauration de l'état courant et retour dans le programme. Toutes ces instructions,

8 - En toute rigueur, l'incorporation est réalisée au niveau du préprocesseur, lequel introduit les instructions en langage C correspondant à "l'expansion" de la macro ; ces instructions peuvent d'ailleurs varier d'un appel à un autre.

9 - Par le compilateur, cette fois, sous forme d'instructions en langage machine.

10 - En toute rigueur, il faudrait plutôt parler de fonction appelante.

nécessaires à la mise en œuvre de l'appel de la fonction, n'existent pas dans le cas de la macro. On peut donc dire que la fonction permet de gagner de l'espace mémoire, en contrepartie d'une perte de temps d'exécution. Bien entendu, la perte de temps sera relativement d'autant plus faible que la fonction sera de taille importante ;

- les macros peuvent, contrairement aux fonctions, entraîner des "effets de bord" indésirables, ou pour le moins, pas nécessairement prévus. Citons deux exemples :
  - si une macro introduit de nouvelles variables, celles-ci peuvent interférer avec d'autres variables de même nom. Ce risque n'existe pas avec une fonction, sauf si l'on utilise, volontairement cette fois, des variables globales ;
  - une macro définie par :

```
carre(x) x * x
```

    et appelée par :

```
carre(a++)
```

    générera les instructions :

```
a++ * a++
```

    qui incrémentent deux fois la variable a.

Généralement, en C, lorsque l'on a besoin d'une fonction courte et que le temps d'exécution est primordial, on fait appel à une macro, malgré les inconvénients que cela implique. En C++, il existe, dans ce cas, une solution plus satisfaisante : utiliser une fonction "en ligne" (inline).

7.2 Utilisation de fonctions "en ligne"

Une fonction "en ligne" se définit et s'utilise comme une fonction ordinaire, avec la seule différence qu'on fait précéder son en-tête de la spécification inline. En voici un exemple.

```
#include <iostream.h>
#include <math.h>
     /* définition d'une fonction "inline" */
inline double norme (double vec[3])
{  int i ; double s = 0 ;
   for (i=0 ; i<3 ; i++)  s+= vec[i]*vec[i] ;
   return sqrt(s) ;
}

        /* exemple d'utilisation */
main()
{  double v1[3], v2[3] ;
   int i ;
   for (i=0 ; i<3 ; i++) { v1[i] = i ; v2[i] = 2*i-1 ; }
   cout << "norme de v1 : " << norme(v1) << " - norme de v2 : " << norme(v2) ;
```

```
}

norme de v1 : 2.236068 - norme de v2 : 3.316625
```

Exemple de définition et d'utilisation d'une fonction "en ligne"

La fonction norme a pour but de calculer la norme d'un vecteur à trois composantes qu'on lui fournit en argument.

La présence du mot inline demande au compilateur de traiter la fonction norme d'une manière différente d'une fonction ordinaire. Plus précisément, à chaque appel de norme, il devra incorporer, au sein du programme, les instructions correspondantes (en langage machine[11]). Le mécanisme habituel de gestion de l'appel et du retour n'existera plus (il n'y a plus besoin de sauvegardes, recopies, ...), ce qui réalise une économie de temps. Par contre, les instructions correspondantes seront générées à chaque appel, ce qui consommera une quantité de mémoire croissant avec le nombre d'appels.

Il est très important de noter que, de par sa nature même, une fonction "en ligne" doit être définie dans le même fichier source que celui où on l'utilise. **Elle ne peut plus être compilée séparément** ! Ceci explique qu'il ne soit pas nécessaire de déclarer une telle fonction (sauf si elle est utilisée, au sein d'un fichier source, avant d'être définie). Ainsi, dans notre exemple, ne trouvons-nous pas de déclaration telle que :

```
double norme (double) ;
```

Cette absence de possibilité de compilation séparée constitue une contrepartie notable aux avantages offerts par la fonction "en ligne". En effet, pour qu'une même fonction "en ligne" puisse être partagée par différents programmes, il faudra absolument la placer dans un fichier en-tête[12] (comme on le fait avec une macro).

	Avantages	Inconvénients
Macro	- Économie de temps d'exécution	- Perte d'espace mémoire - Risque d'effets de bord non désirés - Pas de compilation séparée possible
Fonction	- Économie d'espace mémoire - Compilation séparée possible	- Perte de temps d'exécution
Fonction "en ligne"	- Économie de temps d'exécution	- Perte d'espace mémoire - Pas de compilation séparée possible

Comparaison entre macro, fonction et fonction "en ligne"

11 - Notez qu'il s'agit bien, ici, d'un travail effectué par le compilateur lui-même, alors que dans le cas d'une macro, un travail comparable était effectué par le préprocesseur.

12 - A moins d'en écrire plusieurs fois la définition, ce qui ne serait pas "raisonnable", compte tenu des risques d'erreurs que cela comporte.

Remarque importante :

La déclaration inline constitue une demande effectuée auprès du compilateur. Ce dernier peut éventuellement (par exemple, si la fonction est volumineuse) ne pas l'introduire en ligne et en faire une fonction ordinaire. De même, si vous utilisez quelque part (au sein du fichier source concerné) l'adresse d'une fonction déclarée inline, le compilateur en fera une fonction ordinaire (dans le cas contraire, il serait incapable de lui attribuer une adresse et encore moins de mettre en place un éventuel appel d'une fonction située à cette adresse).

8. LES ESPACES DE NOMS

Lorsque l'on doit utiliser plusieurs bibliothèques d'objets dans un programme, on peut être confronté au problème dit de "pollution de l'espace des noms", lié à ce qu'un même identificateur peut très bien avoir été utilisé par plusieurs bibliothèques. Le même problème peut également se poser, à un degré moindre toutefois, lors du développement de gros programmes. C'est la raison pour laquelle la norme ANSI a introduit le concept d'"espace de noms". Il s'agit simplement de donner un nom à un "espace" de déclarations, en procédant ainsi :

```
namespace une_bibli
{  // déclarations usuelles
}
```

Pour se référer à des identificateurs définis dans cet espace de noms, on utilisera une instruction using :

```
using namespace une_bibli
// ici, les identificateurs de une_bibli sont connus
```

On peut lever l'ambiguïté risquant d'apparaître lorsqu'on utilise plusieurs espaces de noms comportant des identificateurs identiques ; il suffit pour cela de faire appel à l'opérateur de résolution de portée ; par exemple :

```
une_bibli::point ...     // on se réfère à l'identificateur point
                         //   de l'espace de noms une_bibli
```

On peut aussi utiliser l'instruction using pour faire un choix permanent :

```
using une_bibli::point ;    // dorénavant, l'identificateur point, employé seul
                            //   correspondra à celui défini dans
                            //   l'espace de noms une_bibli
```

En particulier, lorsqu'on fera appel aux composants de la bibliothèque standard tels que les conteneurs ou les algorithmes, il faudra mentionner l'instruction :

```
using namespace std ;        /* utilisation de la bibliothèque standard */
```

9. LE TYPE BOOL

Ce type est tout naturellement formé de deux valeurs notées true et false. En théorie, les résultats des comparaisons ou des combinaisons logiques doivent être de ce type. Toutefois, il existe des conversions implicites :

- de bool en numérique, true devenant 1 et false devenant 0,

- de numérique (y compris flottant) en bool, toute valeur non nulle devenant true et zéro devenant false.

Dans ces conditions, tout se passe comme si, finalement, bool était un type énuméré défini ainsi :

```
typedef enum { false=0, true } bool ;
```

En définitive, ce type bool sert surtout à apporter plus de clarté aux programmes, sans remettre en cause quoi que ce soit.

V. CLASSES ET OBJETS

Avec ce chapitre, nous abordons véritablement les possibilités de P.O.O. de C++. Comme nous l'avons dit dans le premier chapitre, celles-ci reposent entièrement sur le concept de classe. Une classe est la généralisation de la notion de type défini par l'utilisateur[1], dans lequel se trouvent associées à la fois des données (membres donnée) et des méthodes (fonctions membre). En P.O.O. "pure", les données sont encapsulées et leur accès ne peut se faire que par le biais des méthodes. C++ vous autorise à n'encapsuler qu'une partie seulement des données d'une classe (cette démarche reste cependant fortement déconseillée). Qui plus est, il existe même un type particulier, correspondant à la généralisation du type structure du C, dans lequel sont effectivement associées des données et des méthodes, mais sans aucune encapsulation.

En pratique, ce nouveau type structure du C++ sera rarement employé sous cette forme généralisée. En revanche, sur le plan conceptuel, il correspond à un cas particulier de la classe ; il s'agit, en effet, d'une classe dans laquelle aucune donnée n'est encapsulée. C'est pour cette raison que nous commencerons par vous présenter le type structure de C++ (mot clé struct) ; ceci nous permettra, dans un premier temps, de nous limiter à la façon de mettre en œuvre l'association des données et des méthodes. Ce n'est qu'ensuite que nous verrons comment s'exprime l'encapsulation au sein d'une classe (mot clé class).

Comme une classe (ou une structure) n'est qu'un simple type défini par l'utilisateur, les objets possèdent les mêmes caractéristiques que les variables ordinaires, en particulier en ce qui concerne leurs différentes classes d'allocation (statique, automatique, dynamique). Cependant, pour rester simple dans un premier temps et nous focaliser essentiellement sur le concept de classe, nous ne considérerons, dans ce chapitre, que des objets automatiques (déclarés au sein d'une fonction quelconque), ce qui correspond au cas le plus naturel. Ce n'est que dans le chapitre VII que nous aborderons les autres classes d'allocation des objets.

Par ailleurs, nous introduirons ici les très importantes notions de constructeur et de destructeur (il n'y a guère d'objets intéressants qui n'y fassent pas appel). Là encore, compte tenu de la richesse de cette notion et de son interférence avec d'autres (comme les classes d'allocation), il vous faudra attendre la fin du chapitre VII pour en connaître toutes les possibilités.

1. LES STRUCTURES EN C++

Rappelons d'abord très succinctement comment on manipule les structures en C.

---

1 - En C, les types définis par l'utilisateur sont : les structures, les unions et les énumérations.

1.1 Rappel : les structures en C

En C, une déclaration telle que :

```
struct point
{   int x ;
    int y ;
} ;
```

définit un "type structure" nommé point (on dit aussi un modèle de structure nommé point ou parfois, par abus de langage, la structure point[2]). Quant à x et y, on dit que ce sont des champs ou des membres[3] de la structure point.

On déclare ensuite des variables du type point par des instructions telles que :

```
struct point a, b ;
```

Celle-ci réserve l'emplacement pour deux structures nommées a et b, de type point. L'accès aux membres (champs) de a ou de b se fait à l'aide de l'opérateur point (.) ; par exemple, a.y désigne le membre y de la structure a.

En C++, comme nous l'avons dit, nous allons pouvoir, dans une structure, associer aux données constituées par ses membres, des méthodes qu'on nommera "fonctions membre". Rappelons que, puisque les données ne sont pas encapsulées dans la structure, une telle association est relativement artificielle et que son principal intérêt est de préparer à la notion de classe.

1.2 Déclaration d'une structure comportant des fonctions membre

Supposez que nous souhaitions associer à la structure point précédente trois fonctions :

- initialise pour donner des valeurs aux "coordonnées" d'un point,
- deplace pour modifier les coordonnées d'un point,
- affiche pour afficher un point : ici, nous nous contenterons, par souci de simplicité, d'afficher les coordonnées du point.

Voici comment nous pourrions déclarer notre structure point.

```
       /* ------------ Déclaration du type point ------------- */
struct point
{              /* déclaration "classique" des données */
    int x ;
    int y ;
          /* déclaration des fonctions membre (méthodes) */
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
```

2 - Dans ce cas, il y a ambiguïté car le même mot structure désignera à la fois un type et des objets d'un type structure. Généralement, le contexte permettra de trancher et c'est souvent ce terme que nous utiliserons.

3 - C'est plutôt ce dernier terme que l'on emploiera en C++.

```
} ;
```

Déclaration d'une structure comportant des méthodes

Nous y trouvons cette fois, outre la déclaration classique des données[4], les déclarations (en-têtes) de nos trois fonctions. Notez bien que la définition de ces fonctions ne figure pas à ce niveau de simple déclaration : elle sera réalisée par ailleurs comme nous le verrons un peu plus loin.

Ici, nous avons prévu que la fonction membre initialise recevra, en argument, deux valeurs de type int. A ce niveau, rien ne dit l'usage qui sera fait de ces deux valeurs. Ici, bien entendu, nous avons écrit l'en-tête de initialise en ayant à l'esprit l'idée qu'elle affecterait aux membres x et y les valeurs reçues en argument. Les mêmes remarques s'appliquent aux deux autres fonctions membre.

Vous vous attendiez (peut-être !) à trouver, pour chaque fonction membre, un argument supplémentaire précisant la structure (variable) sur laquelle elle doit opérer[5]. Nous verrons comment cette information sera automatiquement fournie à la fonction membre lors de son appel.

1.3 Définition des fonctions membre

Elle se fait par une définition (presque) classique de fonction. Voici ce que pourrait être la définition de initialise :

```
void point::initialise (int abs, int ord)
{ x = abs ;
  y = ord ;
}
```

Dans l'en-tête, le nom de la fonction est :

```
point::initialise
```

Le symbole :: correspond à ce que l'on nomme, en C++, l'opérateur de "résolution de portée". Il peut être utilisé dans d'autres contextes que celui-ci ; il sert à modifier la portée d'un identificateur. Ici, il signifie que l'identificateur initialise dont on "parle" est celui défini dans point. En l'absence de ce "préfixe" (point::), nous définirions effectivement une fonction nommée initialise, mais celle-ci ne serait plus associée à point ; il s'agirait d'une fonction "ordinaire", nommée initialise et non plus de la fonction membre initialise de (la structure) point.

Si nous examinons maintenant le corps de la fonction initialise, nous y trouvons une affectation :

```
x = abs ;
```

Le symbole abs y désigne, classiquement, la valeur reçue en premier argument. Mais x, quant à lui, n'est ni un argument ni une variable locale. En fait, x désigne le membre x correspondant au type point (cette association étant réalisée par le point:: de l'en-tête). Quelle sera précisément la structure[6] concernée ? Là

4 - On parle parfois de "variables", par analogie avec les "fonctions membre".

5 - Pour qu'une telle information ne soit pas utile, il faudrait "dupliquer" les fonctions membre en autant d'exemplaires qu'il y a de structures de type point, ce qui serait particulièrement inefficace !

6 - Ici, le terme structure est bien synonyme de variable de type structure.

encore, nous verrons comment cette information sera transmise automatiquement à la fonction initialise lors de son appel.

Nous n'insistons pas sur la définition des deux autres fonctions membre ; vous trouverez ci-dessous l'ensemble des définitions de nos trois fonctions.

```
     /* ----- Définition des fonctions membre du type point ---- */
void point::initialise (int abs, int ord)
{
    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{
    x += dx ; y += dy ;
}
void point::affiche ()
{
    cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Définition des fonctions membre

1.4 Utilisation d'une structure comportant des fonctions membre

Disposant du type point tel qu'il vient d'être déclaré dans le paragraphe 1.2, nous pouvons tout d'abord déclarer autant de structures de ce type que nous le souhaitons. Par exemple :

```
point a, b ;
```
[7]

déclare deux structures nommées a et b, possédant chacune des membres x et y et disposant chacune des trois méthodes initialise, deplace et affiche. A ce propos, nous pouvons d'ores et déjà remarquer que, si chaque structure dispose en propre de chacun de ses membres, il n'en va pas de même des fonctions membre : celles-ci ne sont "générées[8]" qu'une seule fois (le contraire conduirait manifestement à un gaspillage de mémoire !).

L'accès aux membres x et y de nos structures a et b pourrait se dérouler comme en C ; ainsi pourrions-nous écrire :

```
a.x = 5 ;
```

Ce faisant, nous accéderions "directement" aux données, sans passer par l'intermédiaire des méthodes. Certes, nous ne respecterions pas le principe d'encapsulation mais, dans ce cas précis (de structure et pas encore de classe), ce serait parfaitement accepté en C++[9].

L'appel d'une fonction membre est fait d'une manière semblable. Ainsi :

---

7 - Ou struct point a, b ; le mot struct est facultatif en C++.

8 - Exception faite des fonctions "en ligne".

9 - Ici, justement, les fonctions membre prévues pour notre structure point permettent de respecter le principe d'encapsulation.

```
a.initialise (5,2) ;
```

signifie : appeler la fonction membre initialise, **pour la structure a**, en lui transmettant en argument les valeurs 5 et 2. Si l'on fait abstraction du "préfixe" a., cet appel est analogue à un appel classique de fonction. Bien entendu, c'est justement ce préfixe qui va préciser à la fonction membre quelle est la structure sur laquelle elle doit opérer. Ainsi, ici, l'instruction :

```
x = abs ;
```

de point::initialise placera dans le champ x de la structure a, la valeur reçue pour abs (c'est-à-dire 5).

Remarques :

1) Un appel tel que a.initialise (5,2) ; pourrait être remplacé, ici, par :

```
a.x = 5 ; a.y = 2 ;
```

Nous verrons précisément qu'il n'en ira plus de même dans le cas d'une (vraie) classe, pour peu qu'on y ait convenablement encapsulé les données.

2) En jargon P.O.O., on dit également que a.initialise (5, 2) constitue l'**envoi d'un message** (initialise, accompagné des informations 5 et 2) à l'objet a.

1.5 Exemple récapitulatif

Voici un exemple de programme, reprenant la déclaration du type point, la définition de ses fonctions membre et l'utilisant dans la fonction main.

```
#include <iostream.h>
       /* ------------ Déclaration du type point ------------- */
struct point
{              /* déclaration "classique" des données */
    int x ;
    int y ;
          /* déclaration des fonctions membre (méthodes) */
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
     /* ----- Définition des fonctions membre du type point ---- */
void point::initialise (int abs, int ord)
{
    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{
    x += dx ; y += dy ;
}
void point::affiche ()
{
```

```
    cout << "Je suis en " << x << " " << y << "\n" ;
}

main()
{
   point a, b ;
   a.initialise (5, 2) ; a.affiche () ;
   a.deplace (-2, 4) ;  a.affiche () ;
   b.initialise (1,-1) ; b.affiche () ;
}
```

```
Je suis en 5 2
Je suis en 3 6
Je suis en 1 -1
```

Exemple de définition et d'utilisation du type point

Remarques

1) La syntaxe même de l'appel d'une fonction membre fait que celle-ci reçoit obligatoirement un "argument implicite" du type de la structure correspondante. Une fonction membre ne peut pas être appelée comme une "fonction ordinaire". Par exemple, cette instruction :

```
initialise (3,1) ;
```

sera rejetée à la compilation (à moins qu'il n'existe, par ailleurs, une fonction ordinaire nommée initialise).

2) Dans la déclaration d'une structure, il est permis (mais généralement peu conseillé) d'introduire les données et les fonctions dans un ordre quelconque (nous avons systématiquement placé les données avant les fonctions).

3) Dans notre exemple de programme complet, nous avons introduit :

- la déclaration du type point,
- la définition des fonctions membre,
- la fonction (main) utilisant le type point.

Mais, bien entendu, il serait possible de "compiler séparément" le type point ; c'est d'ailleurs ainsi que l'on pourra "réutiliser" un "composant logiciel". Nous y reviendrons à la fin de ce chapitre.

2. NOTION DE CLASSE

Comme nous l'avons déjà dit, la structure en C++ est un cas particulier de la classe. Plus précisément, une classe sera une structure dans laquelle seulement certains membres et/ou fonctions membre seront "publics", c'est-à-dire accessibles "de l'extérieur", les autres membres étant dits "privés".

La déclaration d'une classe est voisine de celle d'une structure puisqu'en effet, il suffit :

- de remplacer le mot clé struct par le mot clé class,
- de préciser quels sont les membres publics (fonctions ou données) et les membres privés en utilisant les mots clés public et private.

Par exemple, faisons de notre précédente structure point une classe dans laquelle tous les membres donnée sont privés et toutes les fonctions membre sont publiques. Sa déclaration serait simplement la suivante.

```
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
  private :          /* facultatif (voir remarque 4) */
    int x ;
    int y ;
               /* déclaration des membres publics */
  public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
```

Déclaration d'une classe

Ici, les membres nommés x et y sont privés, tandis que les fonctions membre nommées initialise, deplace et affiche sont publiques.

En ce qui concerne la définition des fonctions membre d'une classe, elle se fait exactement de la même manière que celle des fonctions membre d'une structure (qu'il s'agisse de fonctions publiques ou privées). En particulier, ces fonctions membre ont (elles !) accès à l'ensemble des membres (publics ou privés) de la classe.

L'utilisation d'une classe se fait également comme celle d'une structure. A titre indicatif, voici ce que devient notre programme du paragraphe 1.5 lorsque l'on y remplace la structure point par la classe point telle que nous venons de la définir.

```
#include <iostream.h>
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
  private :
    int x ;
    int y ;
              /* déclaration des membres publics */
  public :
    void initialise (int, int) ;
```

```
    void deplace (int, int) ;
    void affiche () ;
} ;

     /* ----- Définition des fonctions membre de la classe point ---- */
void point::initialise (int abs, int ord)
{
    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{
    x = x + dx ; y = y + dy ;
}
void point::affiche ()
{
    cout << "Je suis en " << x << " " << y << "\n" ;
}

        /* -------- Utilisation de la classe point -------- */
main()
{
   point a, b ;
   a.initialise (5, 2) ; a.affiche () ;
   a.deplace (-2, 4) ;  a.affiche () ;
   b.initialise (1,-1) ; b.affiche () ;
}
```

Exemple de définition et d'utilisation d'une classe (point)

Remarques

1) Dans le "jargon" de la P.O.O., on dit que a et b sont des ***instances*** de la classe point, ou encore que ce sont des ***objets*** de type point ; c'est généralement ce dernier terme que nous utiliserons[10] ;

2) Dans notre exemple, tous les membres donnée de point sont privés, ce qui réalise une encapsulation complète des données. Ainsi, une tentative d'utilisation directe (ici au sein de la fonction main) du membre a :

   ```
   a.x = 5
   ```

   conduirait à un diagnostic de compilation (bien entendu, cette instruction serait acceptés si nous avions fait de x un membre public).

   En général, on cherchera à respecter le principe d'encapsulation des données, quitte à prévoir des fonctions membre appropriées pour y accéder.

3) Dans notre exemple, toutes les fonctions membre étaient publiques. Il est tout à fait possible d'en rendre certaines privées. Dans ce cas, de telles fonctions ne seront plus accessibles de "l'extérieur" de la classe. Elles ne pourront être appelées que par d'autres fonctions membre.

4) Les mots clés public et private peuvent apparaître à plusieurs reprises dans la définition d'une classe, comme dans cet exemple :

10 - Il pourrait d'ailleurs s'appliquer aux instances de structure

```
class X
{     private :
      ...
      public :
      ...
      private :
      ...
} ;
```

Si aucun de ces deux mots n'apparaît au début de la définition, tout se passe comme si private y avait été placé. C'est pourquoi la présence de ce mot n'était pas indispensable dans la définition de notre classe point.

Si aucun de ces deux mots n'apparaît dans la définition d'une classe, tous ses membres seront donc privés, donc inaccessibles. Cela sera rarement utile.

5) Si l'on rend publics tous les membres d'une classe, on obtient l'équivalent d'une structure. Ainsi, ces deux déclarations définissent le même type point :

```
struct point                class point
{      int x ;                     {   public :
       int y ;                           int x ;
       void initialise (...) ;           int y ;
       .....                       void initialise (...) ;
} ;                                .....
                            } ;
```

6) Par la suite, en l'absence de précisions supplémentaires, nous utiliserons le mot **classe** pour désigner indifféremment une "vraie" classe (**class**) ou une structure (**struct**), voire également une **union** ou une énumération (**enum**) dont nous parlerons un peu plus loin[11]. De même, nous utiliserons le mot **objet** pour désigner des instances de ces différents types.

7) En toute rigueur, il existe un troisième mot, à savoir protected (protégé) qui s'utilise de la même manière que les deux autres ; il sert à définir un statut intermédiaire entre public et privé, lequel n'intervient que dans le cas de classes dérivées. Nous en reparlerons dans le chapitre XII.

3. AFFECTATION D'OBJETS

En C, il est possible d'affecter à une structure la "valeur" d'une autre structure de même type. Ainsi, avec ces déclarations :

```
struct point
{   int x ;
    int y ;
} ;
struct point a, b ;
```

vous pouvez tout à fait écrire :

```
a = b ;
```

Cette instruction recopie l'ensemble des valeurs des champs de b dans ceux de a. Elle joue le même rôle que :

11 - La situation la plus répandue étant celle du type class.

```
a.x = b.x ;
a.y = b.y ;
```

En C++, cette possibilité d'**affectation globale** s'étend aux objets[12] de même type. Elle correspond à une **recopie des valeurs des membres donnée**[13], que ceux-ci soient publics ou non. Ainsi, avec ces déclarations (notez qu'ici nous avons prévu, artificiellement, x privé et y public) :

```
class point
{     int x ;
   public :
      int y ;
      ....
} ;
point a, b ;
```

l'instruction :

```
b = a ;
```

provoquera la recopie des valeurs des membres x et y de a dans les membres correspondants de b.

Il faut noter qu'ici, contrairement à ce qui a été dit pour les structures, il n'est plus possible de remplacer cette instruction par :

```
b.x = a.x ;
b.y = a.y ;
```

En effet, si la deuxième affectation est légale, compte tenu (ici) de ce que y est public, la première ne l'est pas, puisque x est privé[14].

Notez bien que l'affectation b = a est toujours légale, quel que soit le statut (public ou privé) des membres donnée. On peut considérer qu'elle ne viole pas le principe d'encapsulation, dans la mesure où les données privées de b (c'est-à-dire les copies de celles de a) restent toujours inaccessibles de manière directe.

Remarque très importante :

Le rôle de l'opérateur =, tel que nous venons de le définir (recopie des membres données) peut, ici, paraître naturel. En fait, il ne l'est que pour des cas simples. Nous verrons des circonstances où cette banale recopie s'avérera insuffisante ; ce sera notamment le cas dès qu'un objet comportera des pointeurs sur des emplacements dynamiques. La recopie en question ne concernera pas cette partie dynamique de l'objet (on dira qu'il s'agit d'une "recopie superficielle"). Nous reviendrons ultérieurement sur ce point fondamental qui ne trouvera de solution satisfaisante que dans la surdéfinition (pour la classe concernée) de l'opérateur =.

12 - Au sens large : de type class, struct, union ou enum.

13 - Les fonctions membre n'ont aucune raison d'être concernées.

14 - Sauf, si l'affectation b.x = a.x était écrite au sein d'une fonction membre de la classe point.

4. NOTION DE CONSTRUCTEUR ET DE DESTRUCTEUR

4.1 Introduction

A priori, les objets[15] suivent les règles habituelles concernant leur initialisation par défaut, à savoir que seuls les objets statiques voient leurs données initialisées à zéro. En général, il est donc nécessaire de faire appel à une fonction membre pour attribuer des valeurs aux données d'un objet. C'est ce que nous avons fait pour notre type point avec la fonction initialise.

Une telle démarche oblige toutefois à compter sur l'utilisateur de l'objet pour effectuer l'appel voulu au bon moment. En outre, si, dans le cas présent, le risque ne porte que sur des valeurs non définies, il n'en va plus de même dans le cas où, avant même d'être utilisé, un objet doit s'allouer dynamiquement de la mémoire[16]. L'absence de procédure d'initialisation peut, dans ce dernier cas, devenir catastrophique.

C++ offre un mécanisme très performant pour traiter ces problèmes, à savoir le **constructeur**. Il s'agit d'une fonction membre (définie comme les autres fonctions membre) qui sera appelée automatiquement à chaque "création" d'un objet. Ceci aura lieu quelle que soit la "classe d'allocation" de l'objet : statique, automatique ou dynamique. Notez que les objets automatiques auxquels nous nous limitons ici sont "créés" par une déclaration. Ceux de classe dynamique seront créés par new (nous y reviendrons dans le chapitre VII).

D'une manière similaire, un objet pourra posséder un **destructeur** ; il s'agit également d'une fonction membre qui est appelée automatiquement au moment de la "destruction" de l'objet correspondant. Dans le cas des objets automatiques, la destruction de l'objet a lieu lorsque l'on quitte le bloc ou la fonction où il a été déclaré.

Par convention, le constructeur se reconnaît à ce qu'il porte le même nom que la classe. Quant au destructeur, il porte le même nom que la classe, précédé du symbole tilda (~).

4.2 Exemple de classe comportant un constructeur

Considérons notre classe point précédente et transformons simplement notre fonction membre initialise en un constructeur en la renommant point (dans sa déclaration et dans sa définition). Notre nouvelle classe point se présente alors ainsi :

```
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
              /* déclaration des membres publics */
public :
    point (int, int) ;           // constructeur
    void deplace (int, int) ;
    void affiche () ;
} ;
```

15 - Au sens large du terme.

16 - Ne confondez pas un objet dynamique avec un objet (par exemple, automatique) qui s'alloue dynamiquement de la mémoire. Une situation de ce type sera étudiée dans le prochain chapitre.

Une classe (point) munie d'un constructeur

Comment utiliser cette classe ? A priori, vous pourriez penser que la déclaration suivante convient toujours :

```
point a ;
```

En fait, à partir du moment où un constructeur est défini, il doit pouvoir être appelé (automatiquement) lors de la création de l'objet a. Ici, notre constructeur a besoin de deux arguments. Ceux-ci doivent obligatoirement être fournis dans notre déclaration, par exemple :

```
point a(1,3) ;
```

Cette contrainte est en fait un excellent garde-fou : **à partir du moment où une classe possède un constructeur**[17], **il n'est plus possible de créer un objet sans fournir les arguments requis par son constructeur** (sauf si ce dernier ne possède aucun argument[18] !).

A titre d'exemple, voici comment pourrait être adapté le programme du paragraphe 1.5, pour qu'il utilise maintenant notre nouvelle classe point.

```
#include <iostream.h>
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
              /* déclaration des membres publics */
 public :
    point (int, int) ;           // constructeur
    void deplace (int, int) ;
    void affiche () ;
} ;
     /* ----- Définition des fonctions membre de la classe point ---- */
point::point (int abs, int ord)
{    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{    x = x + dx ; y = y + dy ;
}
void point::affiche ()
{    cout << "Je suis en " << x << " " << y << "\n" ;
}
        /* -------- Utilisation de la classe point -------- */
main()
{  point a(5,2) ;
   a.affiche () ;
   a.deplace (-2, 4) ;  a.affiche () ;
```

17 - Nous verrons, dans le prochain chapitre, que le constructeur peut être surdéfini ou posséder des arguments par défaut.

18. Mais, dans tous les cas, il y aura appel du constructeur.

```
   point b(1,-1) ;
   b.affiche () ;
}
Je suis en 5 2
Je suis en 3 6
Je suis en 1 -1
```

Exemple d'utilisation d'une classe (point) munie d'un constructeur

4.3 Construction et destruction des objets

Nous vous proposons ci-dessous un petit programme mettant en évidence les moments où sont appelés respectivement le constructeur et le destructeur d'une classe. Nous y définissons une classe nommée test comportant essentiellement ces deux fonctions membre : celles-ci affichent un message, nous fournissant ainsi une trace de leur appel. En outre, le membre donnée num initialisé par le constructeur nous permet "d'identifier" l'objet concerné (dans la mesure où nous nous sommes arrangés pour qu'aucun des objets créés ne contienne la même valeur). Nous créons des objets automatiques[19] de type test à deux "endroits" différents : dans la fonction main d'une part, dans une fonction fct appelée par main d'autre part.

```
#include <iostream.h>
class test
{
 public :
  int num ;
  test (int) ;           // déclaration constructeur
  ~test () ;             // déclaration destructeur
} ;
test::test (int n)       // définition constructeur
{  num = n ;
   cout << "++ Appel constructeur - num = " << num << "\n" ;
}
test::~test ()           // définition destructeur
{
   cout << "-- Appel destructeur  - num = " << num << "\n" ;
}
main()
{
   void fct (int) ;
   test a(1) ;
   for (int i=1 ; i<=2 ; i++) fct(i) ;
}
void fct (int p)
{   test x(2*p) ;      // notez l'expression (non constante) : 2*p
}
```

```
++ Appel constructeur - num = 1
```

19 - Rappelons qu'ici nous nous limitons à ce cas.

```
++ Appel constructeur - num = 2
-- Appel destructeur  - num = 2
++ Appel constructeur - num = 4
-- Appel destructeur  - num = 4
-- Appel destructeur  - num = 1
```

Construction et destruction des objets

4.4 Rôles du constructeur et du destructeur

Dans nos précédents exemples, le rôle du constructeur se limitait à une "initialisation" de l'objet à l'aide des valeurs qu'il avait reçues en argument. Mais le travail réalisé par le constructeur peut être beaucoup plus élaboré. Voici un programme exploitant une classe nommée hasard, dans laquelle le constructeur fabrique dix valeurs entières aléatoires qu'il range dans le membre donnée val (ces valeurs sont comprises entre zéro et la valeur qui lui est fournie en argument).

```
#include <iostream.h>
#include <stdlib.h>          // pour la fonction rand

class hasard
{  int val[10] ;
 public :
   hasard (int) ;
   void affiche () ;
} ;
hasard::hasard (int max) // constructeur : il tire 10 valeurs au hasard
                         // rappel : rand fournit un entier entre 0 et RAND_MAX
{  int i ;
   for (i=0 ; i<10 ; i++) val[i] = double (rand()) / RAND_MAX * max ;
}
void hasard::affiche ()          // pour afficher les 10 valeurs
{  int i ;
   for (i=0 ; i<10 ; i++) cout << val[i] << " " ;
   cout << "\n" ;
}

main()
{  hasard suite1 (5) ;
   suite1.affiche () ;
   hasard suite2 (12) ;
   suite2.affiche () ;
}
```

```
2 0 1 2 4 0 3 1 2 0
1 4 3 4 11 6 9 7 9 9
```

Un constructeur de valeurs aléatoires

En pratique, d'ailleurs, on préférera disposer d'une classe dans laquelle le nombre de valeurs (ici fixé à dix) peut être fourni en argument du constructeur. Dans ce cas, il est alors préférable que l'espace (variable) soit alloué dynamiquement au lieu d'être surdimensionné. Il est alors tout naturel de faire effectuer cette allocation dynamique par le constructeur lui-même. Les données de la classe hasard se limiteront alors à :

```
class hasard
    {
      int nbval   // nombre de valeurs
      int * val   // pointeur sur un tableau de valeurs
      ...
    } ;
```

Bien sûr, il faudra prévoir que le constructeur reçoive en argument, outre la valeur maximale, le nombre de valeurs souhaitées.

Par ailleurs, à partir du moment où un emplacement a été alloué dynamiquement, il faut se soucier de sa libération lorsqu'il sera devenu inutile. Là encore, il paraît tout naturel de confier ce travail au destructeur de la classe.

Voici comment nous pourrions adapter dans ce sens notre précédent exemple.

```
#include <iostream.h>
#include <stdlib.h>      // pour la fonction rand
class hasard
{
   int nbval ;           // nombre de valeurs
   int * val ;           // pointeur sur les valeurs
 public :
   hasard (int, int) ;   // constructeur
   ~hasard () ;          // destructeur
   void affiche () ;
} ;
hasard::hasard (int nb, int max)
{  int i ;
   val = new int [nbval = nb] ;
   for (i=0 ; i<nb ; i++) val[i] = double (rand()) / RAND_MAX * max ;
}
hasard::~hasard ()
{  delete val ;
}
void hasard::affiche ()          // pour afficher les nbavl valeurs
{  int i ;
   for (i=0 ; i<nbval ; i++) cout << val[i] << " " ;
   cout << "\n" ;
}
main()
{
   hasard suite1 (10, 5) ;       // 10 valeurs entre 0 et 5
   suite1.affiche () ;
   hasard suite2 (6, 12) ;       // 6 valeurs entre 0 et 12
```

```
    suite2.affiche () ;
}
```

```
2 0 1 2 4 0 3 1 2 0
1 4 3 4 11 6
```

Exemple de classe dont le constructeur effectue une allocation dynamique de mémoire

Dans le constructeur, l'instruction :

```
val = new [nbval = nb] ;
```

remplace avantageusement :

```
nbval = nb ;
val = new [nbval] ;
```

Remarques :

1) Ne confondez pas une allocation dynamique effectuée au sein d'une fonction membre d'un objet (souvent le constructeur) avec une allocation dynamique d'un objet, dont nous parlerons plus tard.

2) Lorsqu'un constructeur se contente d'attribuer des valeurs initiales aux données d'un objet, le destructeur est rarement indispensable. Il le devient, par contre, dès que, comme dans notre exemple, l'objet est amené (par le biais de son constructeur ou d'autres fonctions membre) à allouer dynamiquement de la mémoire.

3) Comme nous l'avons déjà mentionné, dès lors qu'une classe contient, comme dans notre dernier exemple, des pointeurs sur des emplacements alloués dynamiquement, l'affectation entre objets de même type ne concerne pas ces parties dynamiques ; généralement, ceci pose problème et la solution passe par la surdéfinition de l'opérateur =. Autrement dit, la classe hasard définie dans le dernier exemple ne permettrait pas de traiter correctement l'affectation d'objets de ce type.

4.5 Quelques règles

Un constructeur peut comporter un nombre quelconque d'arguments, éventuellement aucun. Par définition, un constructeur ne renvoie pas de valeur ; aucun type ne peut figurer devant son nom (la présence de void est, dans ce cas précis, une erreur).

Un destructeur, par définition, ne peut pas disposer d'arguments et ne renvoie pas de valeur. Là encore, aucun type ne peut figurer devant son nom (et la présence de void est une erreur).

En théorie, constructeurs et destructeurs peuvent être publics ou privés. En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il vaut mieux les rendre publics. En effet, un destructeur privé ne pourra pas être appelé directement[20], ce qui n'est pas grave, dans la mesure où cela est rarement utile. En

[20] - L'appel direct d'un destructeur n'est possible que dans la version 2.0.

revanche, si le constructeur d'une classe est privé, il ne sera plus possible de créer des objets de cette classe[21], que ce soit par une déclaration ou une allocation dynamique. Ceci peut, à la rigueur, se justifier, dans le cas où la classe concernée est destinée à donner naissance, par héritage, à des classes dérivées.

5. LES MEMBRES DONNÉE STATIQUES

5.1 Le qualificatif static pour un membre donnée

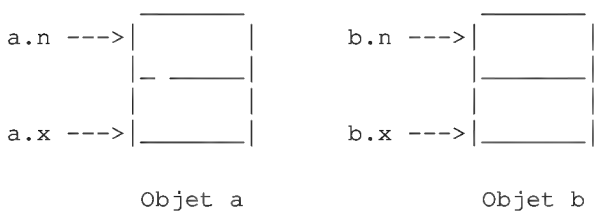
A priori, lorsque dans un même programme on crée différents objets d'une même classe, chaque objet possède ses propres membres donnée. Par exemple, si nous avons défini une classe exple1 par :

```
class exple1
{     int n ;
      float x ;
      ...
} ;
```

Une déclaration telle que :

```
exple1 a, b ;
```

conduit à une situation que l'on peut schématiser ainsi :



Une façon (parmi d'autres) de permettre à plusieurs objets de "partager" des données consiste à déclarer avec le qualificatif static les membres donnée qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe. Par exemple, si nous définissons une classe exple2 par :

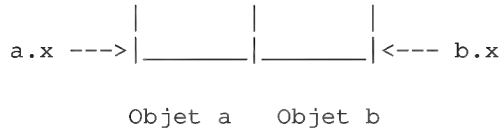
```
class exple2
{     static int n ;
      float x ;
      ...
} ;
```

la déclaration :

```
exple2 a, b ;
```

conduit à une situation que l'on peut schématiser ainsi :

[21] - En utilisant le constructeur en question ; en effet, nous verrons dans le prochain chapitre, qu'une classe peut posséder plusieurs constructeurs (surdéfinis).



On peut dire que les membres donnée statiques sont des sortes de variables globales dont la portée est limitée à la classe.

5.2 Initialisation des membres donnée statiques

De par leur nature même, les membres donnée statiques n'existent qu'en un seul exemplaire, indépendamment des objets de la classe (même, d'ailleurs, si aucun objet de la classe n'a encore été créé). Dans ces conditions, leur initialisation ne peut plus être faite par le constructeur de la classe.

On pourrait penser qu'il est possible d'initialiser un membre statique lors de sa déclaration, comme dans :

```
class exple2
{  static int n = 2 ;      // erreur
   .....
} ;
```

En fait, ceci n'est pas permis car, compte tenu des possibilités de compilation séparée, le membre statique risquerait de se voir réserver différents emplacements[22] dans différents modules objet.

Un membre statique doit donc être initialisé explicitement (à l'extérieur de la déclaration de la classe) par une instruction telle que :

```
int exple2::n = 5 ;
```

Par ailleurs, contrairement à ce qui se produit pour une variable ordinaire, un membre statique n'est pas initialisé par défaut à zéro.

Remarque :

La norme ANSI prévoit qu'on puisse initialiser les membres statiques constants (qualificatif const), lors de leur déclaration.

5.3 Exemple

Voici un exemple de programme exploitant cette possibilité, dans une classe nommée cpte_obj, afin de connaître, à tout moment, le nombre d'objets existant. Pour ce faire, nous avons déclaré avec l'attribut statique le membre ctr. Sa valeur est incrémentée de 1 à chaque appel du constructeur et décrémentée de 1 à chaque appel du destructeur.

```
#include <iostream.h>
```

22 - On retrouve le même phénomène pour les variables globales en langage C : elles peuvent être déclarées plusieurs fois, mais elles ne doivent être définies qu'une seule fois.

```
class cpte_obj
{  static int ctr ;             // compteur du nombre d'objets créés
 public :
   cpte_obj () ;
   ~cpte_obj () ;
} ;
int cpte_obj::ctr = 0 ;  // initialisation du membre statique ctr
cpte_obj::cpte_obj ()           // constructeur
{  cout << "++ construction : il y a maintenant   " << ++ctr << " objets\n" ;
}
cpte_obj::~cpte_obj ()          // destructeur
{   cout << "-- destruction  : il reste maintenant " << --ctr << " objets\n" ;
}
main()
{  void fct () ;
   cpte_obj a ;
   fct () ;
   cpte_obj b ;
}
void fct ()
{  cpte_obj u, v ;
}
```

```
++ construction : il y a maintenant   1 objets
++ construction : il y a maintenant   2 objets
++ construction : il y a maintenant   3 objets
-- destruction  : il reste maintenant 2 objets
-- destruction  : il reste maintenant 1 objets
++ construction : il y a maintenant   2 objets
-- destruction  : il reste maintenant 1 objets
-- destruction  : il reste maintenant 0 objets
```

Exemple d'utilisation de membre statique

Remarques :

1) L'initialisation d'un membre donnée statique est toujours possible suivant la méthode indiquée, qu'il soit public ou privé. En dehors de cela, son accès reste régi comme celui de n'importe quel autre membre de la classe, en fonction de son statut public ou privé.

2) En C, le terme statique avait déjà deux significations : "de classe statique" ou "de portée limitée au fichier source[23]". En C++, lorsqu'il s'applique aux membres d'une classe, il en possède donc une troisième : "indépendant d'une quelconque instance de la classe". Nous verrons, dans le prochain chapitre, qu'il pourra également s'appliquer aux fonctions membre, avec la même signification.

23 - Du moins quand on l'employait pour désigner ce qui était qualifié par le mot clé static.

6. EXPLOITATION D'UNE CLASSE

6.1 La classe, comme "composant logiciel"

Jusqu'ici, nous avions regroupé, au sein d'un même programme, la définition de la classe et son utilisation. Dans la pratique, il en ira généralement autrement. En effet, dans un souci de réutilisabilité, la classe sera fournie comme un composant séparé comme pouvait l'être d'ailleurs une fonction C destinée à être employée par plusieurs programmes (l'exemple le plus flagrant étant celui des fonctions de la bibliothèque standard). Cela signifie qu'un utilisateur potentiel de cette classe disposera :

- d'un module objet résultant de la compilation de la définition de la classe,
- d'un fichier en-tête contenant la déclaration (uniquement) de la classe.

En général, le concepteur de la classe la compilera en utilisant, lui aussi, le fichier en-tête qu'il sera amené à fournir à l'utilisateur. Ce n'est, bien sûr, pas une nécessité, mais cela simplifie d'éventuelles modifications ultérieures de la classe.

Par exemple, le concepteur de la classe point du paragraphe 4.2 pourra créer le fichier en-tête suivant :

```
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
              /* déclaration des membres publics */
public :
    point (int, int) ;           // constructeur
    void deplace (int, int) ;
    void affiche () ;
} ;
```

Fichier en-tête pour la classe point

Si ce fichier se nomme point.h[24], le concepteur fabriquera alors un module objet, en compilant la définition de la classe point.

```
#include <iostream.h>
#include "point.h"
     /* ----- Définition des fonctions membre de la classe point ---- */
point::point (int abs, int ord)
{
   x = abs ; y = ord ;
}
```

[24] - Le nom du fichier peut, bien sûr, être choisi comme vous le voulez. L'extension, par contre, est imposée par l'implémentation. Il peut s'agir de h, de hxx, hpp, ...

```
void point::deplace (int dx, int dy)
{   x = x + dx ; y = y + dy ;
}
void point::affiche ()
{
   cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Fichier à compiler pour obtenir le module objet de la classe point[25]

Pour faire appel à la classe point au sein d'un programme, l'utilisateur procédera alors ainsi :

- il "inclura" la déclaration de la classe point dans le fichier source contenant son programme par une directive telle que :

```
#include "point.h"
```

- il incorporera le module objet correspondant, au moment de l'édition de liens de son propre programme. En principe, à ce niveau, la plupart des éditeurs de liens n'introduisent que les fonctions réellement utilisées, de sorte qu'il ne faut pas craindre de prévoir trop de méthodes pour une classe.

Parfois, on trouvera plusieurs classes différentes au sein d'un même module objet et d'un même fichier en-tête, de façon comparable à ce qui se passe pour les fonctions de la bibliothèque standard[26]. Là encore, en général, seules les fonctions réellement utilisées seront incorporées à l'édition de liens, de sorte qu'il est toujours possible d'effectuer des regroupements de classes possédant quelques affinités.

6.2 Protection contre les inclusions multiples

Plus tard, nous verrons qu'il existe différentes circonstances pouvant amener l'utilisateur d'une classe à inclure plusieurs fois un même fichier en-tête, lors de la compilation d'un même fichier source (sans même qu'il n'en ait conscience !). Ce sera notamment le cas dans les situations d'objets membre et de classes dérivées.

Dans ces conditions, on risque d'aboutir à des erreurs de compilation, liées tout simplement à la redéfinition de la classe concernée.

En général, on réglera ce problème en protégeant systématiquement tout fichier en-tête des inclusions multiples par une technique de compilation conditionnelle, comme dans :

```
#ifndef POINT_H
#define POINT_H
// déclaration de la classe point
#endif
```

Le symbole défini pour chaque fichier en-tête sera choisi de façon à éviter tout risque de doublons. Ici, nous avons choisi le nom de la classe (en majuscules), suffixé par _H.

---

25 - Rappelons que la directive #include possède deux syntaxes différentes, l'une (<...>) effectuant la recherche dans un répertoire spécifique (où se trouvent les fichiers en-tête standard), l'autre ("...") effectuant la recherche dans le répertoire courant ; éventuellement, on peut également y spécifier un répertoire particulier.

26 - Avec cette différence, toutefois, que, dans ce cas, on n'a pas à spécifier les modules objets concernés, au moment de l'édition de liens.

6.3 Cas des membres donnée statiques

Nous avons vu (paragraphe 5.2) qu'un membre donnée statique doit toujours être initialisé explicitement. Dès lors qu'on est amené à considérer une classe comme un composant séparé, le problème se pose alors de savoir dans quel fichier source placer une telle initialisation : fichier en-tête, fichier définition de la classe, fichier utilisateur (dans notre exemple du paragraphe 5.3, ce problème ne se posait pas car nous n'avions qu'un seul fichier source).

On pourrait penser que le fichier en-tête est un excellent candidat pour cette initialisation, dès lors qu'il est protégé contre les inclusions multiples. En fait, il n'en est rien ; en effet, si l'utilisateur compile séparément plusieurs fichiers source utilisant la même classe, plusieurs emplacements seront générés pour le même membre statique et, en principe, l'édition de liens détectera cette erreur.

Comme par ailleurs il n'est guère raisonnable de laisser l'utilisateur initialiser lui-même un membre statique (sauf situations , on voit que :

il est conseillé de prévoir l'initialisation des membres donnée statiques dans le fichier contenant la définition de la classe.

6.4 En cas de modification d'une classe

A priori, lorsqu'une classe est considérée comme un "composant logiciel", c'est qu'elle est "au point" et que, par conséquent, elle ne devrait plus être modifiée. Si, malgré tout, une telle modification s'avère nécessaire, il faut envisager deux situations assez différentes.

a) La déclaration des membres publics n'a pas changé

C'est ce qui se produit lorsqu'on se limite à des modifications internes, n'ayant aucune répercussion sur la manière d'utiliser la classe (son "interface" avec l'extérieur reste la même). Il peut s'agir de transformation de structures de données encapsulées (privées), de modification d'algorithmes de traitement...

Dans ce cas, **les programmes utilisant la classe** n'ont pas à être modifiés. Néanmoins, ils **doivent être recompilés avec le nouveau fichier en-tête correspondant**[27]. On procédera ensuite à une édition de liens en incorporant le nouveau module objet.

On voit donc que C++ permet une maintenance facile d'une classe à laquelle on souhaite apporter des modifications internes (corrections d'erreurs, amélioration des performances...) n'atteignant pas la spécification de son interface.

b) La déclaration des membres publics a changé

Ici, il est clair que les programmes utilisant la classe risquent de nécessiter des modifications. Cette situation devra, bien sûr, être évitée dans la mesure du possible. Elle doit être considérée comme une faute de conception de la classe. Nous verrons d'ailleurs que ces problèmes pourront souvent être résolus par l'emploi du mécanisme d'héritage qui permet d'adapter une classe sans la remettre en cause.

---

[27] - Une telle limitation n'existe pas dans tous les langages de P.O.O. En C++, elle se justifie par le besoin qu'a le compilateur de connaître la taille des objets pour leur allouer un emplacement.

7. LES CLASSES EN GÉNÉRAL

7.1 Les autres sortes de classes en C++

Nous avons déjà eu l'occasion de dire que C++ qualifiait de "classe" les types définis par struct et class. La caractéristique d'une classe, au sens large que lui donne C++[28], est d'associer, au sein d'un même type, des membres donnée et des fonctions membre.

Pour C++, les **unions** et les **énumérations sont aussi des classes**. Ces deux types peuvent donc disposer de fonctions membre. Notez bien que, comme pour le type struct, les données correspondantes ne peuvent pas se voir attribuer un "statut" particulier : elles sont, de fait, publiques.

7.2 Ce qu'on peut trouver dans la déclaration d'une classe

En dehors des déclarations de fonctions membre, la plupart des instructions figurant dans une déclaration de classe seront des déclarations de membres donnée d'un type quelconque. Néanmoins, on peut y rencontrer également des déclarations de type, y compris d'autres types classes ; dans ce cas, leur portée est limitée à la classe. En pratique, cette situation se rencontre peu souvent.

Par ailleurs, il **n'est pas possible d'initialiser un membre donnée** d'une classe, lors de sa déclaration[29]. Cette interdiction est justifiée pour au moins deux raisons :

- une telle initialisation risquerait de faire double emploi avec le constructeur,
- une telle initialisation constituerait une définition du membre correspondant (et non plus une simple déclaration) ; or, cette définition risquerait d'apparaître plusieurs fois en cas de compilation séparée, ce qui est illégal[30].

En revanche, la déclaration de membres donnée constants[31] est autorisée, comme dans :

```
class exple
{  int n ;           // membre donnée usuel
   const int p ;     // membre donnée constant
   .....
} ;
```

Dans ce cas, cependant, il n'est pas permis d'initialiser le membre constant au moment de sa déclaration. Pour y parvenir, la seule solution consistera à utiliser une "syntaxe" particulière du constructeur, telle qu'elle sera présentée au chapitre suivant, dans le paragraphe relatif aux "objets membre".

---

28 - Et non la P.O.O. d'une manière générale qui associe l'encapsulation des données à la notion de classe.

29 - Mais, comme nous l'avons dit dans le paragraphe 5.2, la norme ANSI fait une exception pour les membres statiques constants.

30 - On retrouve le même phénomène pour les membres données statiques et pour les variables globales en langage C : ils peuvent être déclarés plusieurs fois, mais ils ne doivent être définis qu'une seule fois.

31 - Ne confondez pas la notion de membre donnée constant (chaque objet en possède un ; sa valeur ne peut pas être modifiée) et la notion de membre donnée statique (tous les objets d'une même classe partagent le même ; sa valeur peut changer).

7.3 Déclaration d'une classe

La plupart du temps, les classes seront déclarées à un niveau global. Néanmoins, il est permis de déclarer des classes locales à une fonction. Dans ce cas, leur portée est naturellement limitée à cette fonction (c'est bien ce qui en limite l'intérêt).

EXERCICES

N.B : les exercices marqués (C) sont corrigés en fin de volume.

1 - Expérimentez (éventuellement sur un exemple de ce chapitre) la compilation séparée d'une classe (création d'un module objet et d'un fichier en-tête) et son utilisation au sein d'un programme.

2 - (C) Ecrivez une classe vecteur (de type class et non struct) comportant :

- en membres donnée privés : trois composantes de type double,
- en fonctions membre publiques :
  * initialise pour attribuer des valeurs aux composantes,
  * homothetie pour multiplier les composantes par une valeur fournie en argument,
  * affiche pour afficher les composantes du vecteur.

3 - (C) Ecrivez une classe vecteur, analogue à la précédente, dans laquelle la fonction initialise est remplacée par un constructeur.

4 - Expérimentez la création d'un fichier en-tête et d'un module objet rassemblant deux classes différentes.

5 - Vérifiez que, lorsqu'une classe comporte un membre donnée statique, ce dernier peut être utilisé, même lorsqu'aucun objet de ce type n'a été déclaré.

6 - Mettez en évidence les problèmes posés par l'affectation entre objets comportant une partie dynamique. Pour ce faire, utilisez la classe hasard du second exemple du paragraphe 4.4, en ajoutant simplement des instructions affichant l'adresse contenue dans val, dans le constructeur d'une part, dans le destructeur d'autre part. Vous constaterez qu'avec ces déclarations :

```
hasard h1(10, 3) ;
hasard h2(20, 5) ;
```

une instruction telle que :

```
h2 = h1 ;
```

n'entraîne pas toutes les recopies escomptées et que, de surcroît, elle conduit à libérer (en fin de fonction) deux fois le même emplacement.

VI. LES PROPRIÉTÉS DES FONCTIONS MEMBRE

Le chapitre précédent vous a présenté les concepts fondamentaux de classe, d'objet, de constructeur et de destructeur. Ici, nous allons étudier un peu plus en détail l'application aux fonctions membre des possibilités offertes par C++ pour les fonctions ordinaires : surdéfinition, arguments par défaut, fonction "en ligne", transmission par référence.

Nous verrons également comment une fonction membre peut recevoir en argument, outre l'objet l'ayant appelé (transmis implicitement) un ou plusieurs objets de type classe. Ici, nous nous limiterons au cas d'objets de même type que la classe dont la fonction est membre ; les autres situations, correspondant à une violation du principe d'encapsulation, ne seront examinées que plus tard, dans le cadre des "fonctions amies".

Nous verrons ensuite comment, au sein d'une fonction membre, accéder à l'adresse de l'objet l'ayant appelé, en utilisant le mot clé this.

Enfin, nous examinerons les cas particuliers des fonctions membre statiques et des fonctions membre constantes, ainsi que l'emploi de pointeurs sur des fonctions membre.

1. SURDÉFINITION DES FONCTIONS MEMBRE

Nous avons déjà vu comment C++ nous autorise à "surdéfinir" les fonctions "ordinaires". Cette possibilité s'applique également aux fonctions membre d'une classe, y compris au constructeur lui-même (mais pas au destructeur puisqu'il ne possède pas d'arguments). En voici un exemple, dans lequel nous surdéfinissons :

- le constructeur point ; le choix du bon constructeur se faisant (ici) suivant le nombre d'arguments :
  - 0 argument : les deux coordonnées attribuées au point construit sont toutes deux nulles,
  - 1 argument : il sert de valeur commune aux deux coordonnées,
  - 2 arguments : c'est le cas "usuel" que nous avions déjà rencontré.
- la fonction affiche de manière qu'on puisse l'appeler :
  - sans argument comme auparavant,

- avec un argument de type chaîne : dans ce cas, elle affiche le texte correspondant avant les coordonnées du point.

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point () ;                // constructeur 1 (sans arguments)
   point (int) ;             // constructeur 2 (un argument)
   point (int, int) ;        // constructeur 3 (deux arguments)
   void affiche () ;         // fonction affiche 1 (sans arguments)
   void affiche (char *) ;   // fonction affiche 2 (un argument chaîne)
} ;
point::point ()                                  // constructeur 1
{  x = 0 ; y = 0 ;
}
point::point (int abs)                           // constructeur 2
{  x = y = abs ;
}
point::point (int abs, int ord)                  // constructeur 3
{  x = abs ; y = ord ;
}
void point::affiche ()                           // fonction affiche 1
{  cout << "Je suis en : " << x << " " << y << "\n" ;
}
void point::affiche (char * message)             // fonction affiche 2
{  cout << message ; affiche () ;
}
main()
{
   point a ;                     // appel constructeur 1
   a.affiche () ;                // appel fonction affiche 1
   point b (5) ;                 // appel constructeur 2
   b.affiche ("Point b - ") ;    // appel fonction affiche 2
   point c (3, 12) ;             // appel constructeur 3
   c.affiche ("Hello ---- ") ;   // appel fonction affiche 2
}
```

```
Je suis en : 0 0
Point b - Je suis en : 5 5
Hello ---- Je suis en : 3 12
```

Exemple de surdéfinition de fonctions membre (point et affiche)

Remarques :

1) Souvent, en utilisant les possibilités d'arguments par défaut, il est possible de diminuer le nombre de fonctions surdéfinies. C'est le cas ici pour la fonction affiche, comme nous le verrons d'ailleurs dans le paragraphe suivant.

2) Ici, dans la fonction affiche(char *), nous faisons appel à l'autre fonction membre affiche(). En effet, une fonction membre peut toujours en appeler une autre (qu'elle soit publique ou non). Une fonction membre peut même s'appeler elle-même, dans la mesure où l'on a prévu le moyen de rendre fini le processus de récursivité qui en découle.

2. ARGUMENTS PAR DÉFAUT

Les fonctions membre, comme les fonctions "ordinaires" peuvent disposer d'arguments par défaut. Voici comment nous pourrions modifier l'exemple précédent pour que notre classe point ne possède plus qu'une seule fonction affiche à un seul argument de type chaîne : le message à afficher avant les valeurs des coordonnées, sa valeur par défaut étant alors la chaîne vide.

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
   point () ;                    // constructeur 1 (sans argument)
   point (int) ;                 // constructeur 2 (un argument)
   point (int, int) ;            // constructeur 3 (deux arguments)
   void affiche (char * = "") ;  // fonction affiche (un argument par défaut)
} ;
point::point ()                          // constructeur 1
{  x = 0 ; y = 0 ;
}
point::point (int abs)                   // constructeur 2
{  x = y = abs ;
}
point::point (int abs, int ord)          // constructeur 3
{  x = abs ; y = ord ;
}
void point::affiche (char * message)     // fonction affiche
{  cout << message << "Je suis en : " << x << " " << y << "\n" ;
}
main()
{  point a ;                        // appel constructeur 1
   a.affiche () ;
   point b (5) ;                    // appel constructeur 2
   b.affiche ("Point b - ") ;
   point c (3, 12) ;                // appel constructeur 3
   c.affiche ("Hello ---- ") ;
}
```

```
Je suis en : 0 0
Point b - Je suis en : 5 5
Hello ---- Je suis en : 3 12
```

Exemple d'utilisation d'arguments par défaut dans une fonction membre

Remarque :

Ici, nous avons remplacé deux fonctions surdéfinies par une seule fonction ayant un argument par défaut. Bien entendu, cette simplification n'est pas toujours possible. Par exemple, ici, nous ne pouvons pas l'appliquer à notre constructeur point. Si, par contre, nous avions prévu que, dans le constructeur point à un seul argument, ce dernier représente simplement l'abscisse du point auquel on aurait alors attribué une ordonnée nulle, nous aurions pu alors définir un seul constructeur :

```
point::point (int abs = 0, int ord = 0)
       {x = abs ; y = ord ;}
```

3. LES FONCTIONS MEMBRE "EN LIGNE"

Nous avons vu que C++ permet de définir des fonctions "en ligne". Ceci accroît l'efficience d'un programme, dans le cas de fonctions courtes. Là encore, cette possibilité s'applique aux fonctions membre, moyennant cependant une petite nuance concernant sa mise en œuvre. En effet, pour rendre "en ligne" une fonction membre, on peut :

- soit fournir directement la définition de la fonction dans la déclaration même de la classe ; dans ce cas le qualificatif inline n'a pas à être utilisé,
- soit procéder comme pour une fonction "ordinaire" en fournissant une définition en dehors de la déclaration de la classe ; dans ce cas, le qualificatif inline doit apparaître, à la fois devant la déclaration et devant l'en-tête.

Voici comment nous pourrions rendre "en ligne" les trois constructeurs de notre précédent exemple en adoptant ici la première manière.

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
   point () { x = 0 ; y = 0 ; }                      // constructeur 1 "en
ligne"
   point (int abs) { x = y = abs ; }                 // constructeur 2 "en
ligne"
   point (int abs, int ord) { x = abs ; y = ord ; }  // constructeur 3 "en
ligne"
   void affiche (char * = "") ;
} ;

void point::affiche (char * message)                 // fonction affiche
{  cout << message << "Je suis en : " << x << " " << y << "\n" ;
}

main()
{
   point a ;                      // "appel" constructeur 1
   a.affiche () ;
   point b (5) ;                  // "appel" constructeur 2
```

```
    b.affiche ("Point b - ") ;
    point c (3, 12) ;              // "appel" constructeur 3
    c.affiche ("Hello ---- ") ;
}
```

```
Je suis en : 0 0
Point b - Je suis en : 5 5
Hello ---- Je suis en : 3 12
```

Exemple de fonctions membre "en ligne"

Remarques

1) Voici comment se serait présentée la déclaration de notre classe si nous avions déclaré nos fonctions membre en ligne à la manière des fonctions ordinaires (ici, nous n'avons mentionné qu'un constructeur) :

```
class point
{  .....
 public :
  inline point () ;
    .....
} ;
inline point::point()  { x = 0 ; y = 0 ; }
  .....
```

2) Si nous n'avions eu besoin que d'un seul constructeur avec arguments par défaut (comme dans la remarque du précédent paragraphe), nous aurions pu tout aussi bien le rendre "en ligne" ; avec la première démarche (définition de fonction intégrée dans la déclaration de la classe), nous aurions alors spécifié les valeurs par défaut directement dans l'en-tête :

```
class point
{      ...
       point (int abs = 0, int ord = 0)
             {x = abs ; y = ord ;}
```

Nous utiliserons d'ailleurs un tel constructeur dans l'exemple du paragraphe suivant.

3) De par sa nature même, la définition d'une fonction "en ligne" doit obligatoirement être connue du compilateur lorsqu'il traduit le programme qui l'utilise. Cette condition est obligatoirement réalisée lorsque l'on utilise la première démarche. En revanche, ce n'est plus vrai avec la deuxième démarche ; en général, dans ce cas, on placera les définitions des fonctions en ligne, à la suite de la déclaration de la classe, dans le même fichier en-tête.

Dans tous les cas, on voit toutefois que l'utilisateur d'une classe (qui disposera obligatoirement du fichier en-tête relatif à une classe) pourra toujours connaître la définition des fonctions en ligne ; le fournisseur d'une classe ne pourra jamais avoir la certitude qu'un utilisateur de cette classe ne tentera pas de les modifier. Ce risque n'existe pas pour les autres fonctions membre (dès lors que l'utilisateur ne dispose que du module objet relatif à la classe).

4. CAS DES OBJETS TRANSMIS EN ARGUMENT D'UNE FONCTION MEMBRE

Dans nos précédents exemples, les fonctions membre recevaient :

- un argument implicite du type de leur classe, à savoir l'adresse de l'objet l'ayant appelé,
- un certain nombre d'arguments qui étaient d'un type "ordinaire" (c'est-à-dire autre que classe).

Mais une fonction membre peut, outre l'argument implicite, recevoir un ou plusieurs arguments du type de sa classe. Par exemple, supposez que nous souhaitions, au sein d'une classe point, introduire une fonction membre nommée coincide, chargée de détecter la coïncidence éventuelle de deux points. Son appel, au sein d'un programme, se présentera obligatoirement, comme pour toute fonction membre, sous la forme :

```
a.coincide (...)
```

a étant un objet de type point.

Il faudra donc impérativement transmettre le second point en argument ; en supposant qu'il se nomme b, ceci nous conduira à un appel de la forme :

```
a.coincide (b)
```

ou, ici, compte tenu de la "symétrie" du problème :

```
b.coincide (a)
```

Voyons maintenant plus précisément comment écrire la fonction coincide. Voici ce que peut être son en-tête, en supposant qu'elle fournit une valeur de retour entière (1 en cas de coïncidence, 0 dans le cas contraire) :

```
int point::coincide (point pt)
```

Dans coincide, nous devons donc comparer les coordonnées de l'objet fourni implicitement lors de son appel (ses membres sont désignés, comme d'habitude, par x et y) avec les coordonnées de l'objet fourni en argument, dont les membres sont désignés par pt.x et pt.y. Le corps de coincide se présentera donc ainsi :

```
if ((pt.x == x) && (pt.y == y)) return 1 ;
                           else return 0 ;
```

Voici un exemple complet de programme, dans lequel nous avons limité les fonctions membre de la classe point à un constructeur et à coincide.

```
#include <iostream.h>
class point                     // Une classe point contenant seulement :
{
   int x, y ;
 public :
   point (int abs=0, int ord=0)      // un constructeur ("en ligne")
       { x=abs; y=ord ; }
   int coincide (point) ;            // une fonction membre : coincide
} ;

int point::coincide (point pt)
{  if ( (pt.x == x) && (pt.y == y) ) return 1 ;
                                else return 0 ;
```

```
                        //  remarquez la "dissymétrie" des notations : pt.x et x
}

main()                                  // Un petit programme d'essai
{
   point a, b(1), c(1,0) ;
   cout << "a et b : " << a.coincide(b) << " ou " << b.coincide(a) << "\n" ;
   cout << "b et c : " << b.coincide(c) << " ou " << c.coincide(b) << "\n" ;
}
```

```
a et b : 0 ou 0
b et c : 1 ou 1
```

Exemple d'objet transmis en argument à une fonction membre

Remarques

1) Nous aurions pu écrire coincide de la manière suivante

```
return ((pt.x == x) && (pt.y == y)) ;
```

2) En théorie, on peut dire que la coïncidence de deux points est "symétrique", en ce sens que l'ordre dans lequel on considère les deux points est indifférent. Or, cette symétrie ne se retrouve pas dans la définition de la fonction coincide, pas plus que dans son appel. Ceci provient de la transmission, en argument implicite, de l'objet appelant la fonction.

3) On pourrait penser qu'on viole le principe d'encapsulation dans la mesure où, lorsque l'on appelle la fonction coincide pour l'objet a (dans a.coincide(b)), elle est autorisée à accéder aux données de b. En fait, en C++, n'importe quelle fonction membre d'une classe peut accéder à n'importe quel membre (public ou privé) de n'importe quel objet de cette classe. On traduit souvent cela en disant qu'**en C++, l'unité de protection est la classe**, et non l'objet.

   En revanche, si A et B sont deux classes différentes, une fonction membre de A ne peut pas (pas plus qu'une fonction ordinaire, main par exemple) accéder aux membres privés d'un objet de classe B ; bien entendu, elle peut toujours accéder aux membres publics. Nous verrons plus tard qu'il est possible à une fonction (ordinaire ou membre) de s'affranchir de cette interdiction (et donc, cette fois, de violer véritablement le principe d'encapsulation) par des déclarations d'amitié appropriées.

5. MODE DE TRANSMISSION DES OBJETS EN ARGUMENT

Dans notre précédent exemple, l'objet pt était transmis classiquement à coincide, à savoir par valeur. Précisément, cela signifie donc que, lors de l'appel :

```
a.coincide (b)
```

les valeurs des données de b sont recopiées dans un emplacement (de type point) local à coincide (nommé, en quelque sorte, pt).

Comme pour n'importe quel argument ordinaire, il est possible de prévoir d'en transmettre l'adresse plutôt que la valeur ou de mettre en place une transmission par référence. Examinons ces deux possibilités.

5.1 Transmission de l'adresse d'un objet

Il est possible de transmettre explicitement en argument l'adresse d'un objet. Rappelons que, dans un tel cas, on ne change pas le mode de transmission de l'argument (contrairement à ce qui se produit avec la transmission par référence) ; on se contente de transmettre une valeur qui se trouve être une adresse et qu'il faut donc interpréter en conséquence dans la fonction (notamment en employant l'opérateur d'indirection *). A titre d'exemple, voici comment nous pourrions modifier la fonction coincide du paragraphe précédent :

```
int point::coincide (point * adpt)
{   if (( adpt -> x == x) && (adpt -> y == y)) return 1 ;
                                        else return 0 ;
}
```

Compte tenu de la dissymétrie naturelle de notre fonction membre, cette écriture n'est guère choquante. Par contre, l'appel de coincide (au sein de main) le devient davantage :

```
a.coincide (&b)
```

ou

```
b.coincide (&a)
```

Voici le programme complet ainsi modifié.

```
#include <iostream.h>
class point                     // Une classe point contenant seulement :
{  int x, y ;
 public :
   point (int abs=0, int ord=0)       // un constructeur ("en ligne")
       { x=abs; y=ord ; }
   int coincide (point *) ;           // une fonction membre : coincide
} ;
int point::coincide (point * adpt)
{  if ( (adpt->x == x) && (adpt->y == y) ) return 1 ;
                                       else return 0 ;
}

main()                                // Un petit programme d'essai
{  point a, b(1), c(1,0) ;
   cout << "a et b : " << a.coincide(&b) << " ou " << b.coincide(&a) << "\n" ;
   cout << "b et c : " << b.coincide(&c) << " ou " << c.coincide(&b) << "\n" ;
}
```

```
a et b : 0 ou 0
b et c : 1 ou 1
```

Exemple de transmission de l'adresse d'un objet à une fonction membre

Remarque

N'oubliez pas qu'à partir du moment où vous fournissez l'adresse d'un objet à une fonction membre, celle-ci peut en modifier les valeurs (elle a accès à tous les membres s'il s'agit d'un objet de type de sa classe, aux seuls membres publics dans le cas contraire). Si vous craignez de tels effets de bord au sein de la fonction membre concernée, vous pouvez toujours employer le qualificatif const. Ainsi, ici, l'en-tête de coincide aurait pu être :

```
int point::coincide (const point * adpt)
```

en modifiant parallèlement son prototype :

```
int coincide (const point *) ;
```

Notez toutefois qu'une telle précaution ne peut pas être prise avec l'argument implicite qu'est l'objet ayant appelé la fonction. Ainsi, dans coincide muni de l'en-tête ci-dessus, vous ne pourriez plus modifier adpt -> x mais vous pourriez toujours modifier x.

5.2 Transmission par référence

Comme nous l'avons vu, l'emploi des références permet de mettre en place une transmission par adresse, sans avoir à en prendre en charge soi-même la gestion. Elle simplifie d'autant l'écriture de la fonction concernée et ses différents appels. Voici une adaptation de coincide dans laquelle son argument est transmis par référence.

```
#include <iostream.h>
class point                     // Une classe point contenant seulement :
{  int x, y ;
 public :
   point (int abs=0, int ord=0)       // un constructeur ("en ligne")
       { x=abs; y=ord ; }
   int coincide (point &) ;           // une fonction membre : coincide
} ;
int point::coincide (point & pt)
{  if ( (pt.x == x) && (pt.y == y) ) return 1 ;
                                   else return 0 ;
}

main()                                 // Un petit programme d'essai
{
   point a, b(1), c(1,0) ;
   cout << "a et b : " << a.coincide(b) << " ou " << b.coincide(a) << "\n" ;
   cout << "b et c : " << b.coincide(c) << " ou " << c.coincide(b) << "\n" ;
}
```

```
a et b : 0 ou 0
b et c : 1 ou 1
```

Remarque

La remarque faite précédemment (en fin de paragraphe 5.1) à propos des risques d'effets de bord s'applique également ici. Le qualificatif const pourrait y intervenir de manière analogue :

```
int point::coincide (const point & pt)
```

5.3 Les problèmes posés par la transmission par valeur

Nous avons déjà vu que l'affectation d'objets pouvait poser des problèmes dans le cas où ces objets possédaient des pointeurs sur des emplacements alloués dynamiquement. Ces pointeurs étaient effectivement recopiés, mais il n'en allait pas de même des emplacements pointés. Le transfert d'arguments par valeur présente les mêmes risques, dans la mesure où il s'agit également d'une simple recopie.

De même que le problème posé par l'affectation peut être résolu par la surdéfinition de cet opérateur, celui posé par le transfert par valeur peut être réglé par l'emploi d'un constructeur particulier ; nous vous montrerons comment dès le prochain chapitre.

D'une manière générale, d'ailleurs, nous verrons que les problèmes posés par les objets contenant des pointeurs se ramènent effectivement à l'*affectation* et à l'*initialisation*[1], dont la recopie en cas de transmission par valeur constitue un cas particulier.

6. LORSQUE LA VALEUR DE RETOUR D'UNE FONCTION EST ELLE-MÊME UN OBJET

Ce que nous avons dit à propos des arguments d'une fonction membre s'applique également à sa valeur de retour. Il peut s'agit d'un objet et on peut choisir entre :

- transmission par valeur,
- transmission de son adresse,
- transmission par référence.

Cet objet pourra être :

- du même type que la classe, auquel cas la fonction aura accès à ses membres privés,
- d'un type différent de la classe, auquel cas la fonction n'aura accès qu'à ses membres publics.

La transmission par valeur attire la même remarque que précédemment, à savoir que, par défaut, elle se fait par simple recopie de l'objet. Pour les objets comportant des pointeurs sur des emplacements dynamiques, il faudra prévoir un constructeur particulier (d'initialisation).

En revanche, la transmission d'une adresse ou la transmission par référence risquent de poser un problème qui n'existait pas pour les arguments. Si une fonction transmet l'adresse ou la référence d'un objet, il vaut mieux éviter qu'il s'agisse d'un objet local à la fonction, c'est-à-dire de classe automatique. En effet, dans ce cas, l'emplacement de cet objet sera libéré[2] dès la sortie de la fonction ; la fonction appelante récupérera

---

1 - Bien que cela n'apparaisse pas toujours clairement en C, il est très important, en C++, de noter qu'affectation et initialisation sont deux choses différentes.

2 - Comme nous le verrons en détail dans le chapitre suivant, il y aura appel du destructeur, s'il existe.

l'adresse de quelque chose n'existant plus vraiment[3]. Nous reviendrons plus en détail sur ce point dans le chapitre consacré à la surdéfinition d'opérateurs.

A titre d'exemple, voici une fonction membre nommée symetrique qui pourrait être introduite dans une classe point pour fournir en retour un point symétrique de celui l'ayant appelé :

```
point point::symetrique ( )
{    point res ;
     res.x = -x ; res.y = -y ;
     return res ;
}
```

Vous constatez qu'il a été nécessaire de créer un objet automatique res, au sein de la fonction. Comme nous l'avons expliqué ci-dessus, il ne serait pas conseillé d'en prévoir ici une transmission par référence, en utilisant cet en-tête :

```
point & point::symetrique ( )
```

7. AUTO RÉFÉRENCE : LE MOT CLÉ THIS

Nous avons déjà eu souvent l'occasion de dire qu'une fonction membre d'une classe reçoit une information lui permettant d'accéder à l'objet l'ayant appelé. Ce terme "information", bien qu'il soit relativement flou, nous avait suffi pour expliquer tous les exemples rencontrés jusqu'ici. Mais, nous n'avions pas besoin d'y manipuler explicitement l'adresse de l'objet en question. Or, il existe des circonstances où cela devient indispensable. Songez, par exemple, à la gestion d'une liste chaînée d'objets de même nature : pour écrire une fonction membre insérant un nouvel objet (supposé transmis en argument implicite), il faudra bien placer son adresse dans l'objet précédent de la liste.

Pour résoudre de tels problèmes, C++ a prévu le mot clé : **this**.

Celui-ci, utilisable uniquement au sein d'une fonction membre, désigne un pointeur sur l'objet l'ayant appelé.

Ici, il serait prématuré de développer l'exemple de liste chaînée dont nous venons de parler ; nous vous proposons un exemple d'école : dans la classe point, la fonction affiche fournit l'adresse de l'objet l'ayant appelé.

```
#include <iostream.h>
class point                    // Une classe point contenant seulement :
{
   int x, y ;
 public :
   point (int abs=0, int ord=0)      // Un constructeur ("inline")
       { x=abs; y=ord ; }
   void affiche () ;                 // Une fonction affiche
} ;
void point::affiche ()
```

3 - Dans certaines implémentations, un emplacement libéré n'est pas remis à zéro. Ainsi, on peut avoir l'illusion que "cela marche" si l'on se contente d'exploiter l'objet immédiatement après l'appel de la fonction.

```
{  cout << "Adresse : " << this << " - Coordonnées " << x << " " << y << "\n" ;
}

main()                                  // Un petit programme d'essai
{
   point a(5), b(3,15) ;
   a.affiche ();
   b.affiche ();
}
```

```
Adresse : 0x40d80ffc - Coordonnées 5 0
Adresse : 0x40d80ff8 - Coordonnées 3 15
```

Exemple d'utilisation de this[4]

Remarque :

A titre purement indicatif, la fonction coincide du paragraphe 5.1 pourrait s'écrire :

```
int point::coincide (point * adpt)
{   if ((this -> x == adpt -> x) && (this -> y == adpt -> y)) return 1 ;
                                                              else return 0 ;
}
```

La symétrie du problème y apparaît plus clairement. Ce serait moins le cas si l'on écrivait ainsi la fonction coincide du paragraphe 4 :

```
int point::coincide (point pt)
{   if ((this -> x == pt.x)) && (this -> y == pt.y)) return 1 ;
                                                     else return 0 ;
}
```

8. LES FONCTIONS MEMBRE STATIQUES

Nous avons déjà vu (chapitre V, paragraphe 5) comment C++ permet de définir des membres donnée statiques. Ceux-ci existent en un seul exemplaire (pour une classe donnée), indépendamment des objets de leur classe.

D'une manière analogue, on peut imaginer que certaines fonctions membre d'une classe aient un rôle totalement indépendant d'un quelconque objet ; ce serait notamment le cas d'une fonction qui se contenterait d'agir sur des membres donnée statiques.

On peut certes toujours appeler une telle fonction en la faisant porter artificiellement sur un objet de la classe, et ceci, bien que l'adresse de cet objet ne soit absolument pas utile à la fonction. En fait, il est possible de rendre les choses plus lisibles et plus efficaces, en déclarant statique (mot clé static) la fonction membre concernée. Dans ce cas, en effet, son appel ne nécessite plus que le nom de la classe correspondante (accompagné, naturellement, de l'opérateur de résolution de portée). Comme pour les membres statiques, une telle fonction membre statique peut même être appelée lorsqu'il n'existe aucun objet de sa classe.

4 - Réalisé dans un environnement dans lequel les pointeurs (near) s'expriment sur seize bits.

Voici un exemple de programme illustrant l'emploi d'une fonction membre statique : il s'agit de l'exemple du paragraphe 5.3 du chapitre V, dans lequel nous avons introduit une fonction membre statique nommée compte, affichant simplement le nombre d'objets de sa classe.

```
#include <iostream.h>
class cpte_obj
{  static int ctr ;          // compteur (statique) du nombre d'objets créés
 public :
   cpte_obj () ;
   ~cpte_obj() ;
   static void compte () ;   // pour afficher le nombre d'objets créés
} ;
int cpte_obj::ctr = 0 ;      // initialisation du membre statique ctr
cpte_obj::cpte_obj ()        // constructeur
{
   cout << "++ construction : il y a maintenant   " << ++ctr << " objets\n" ;
}
cpte_obj::~cpte_obj ()       // destructeur
{
   cout << "-- destruction  : il reste maintenant " << --ctr << " objets\n" ;
}
void cpte_obj::compte ()
{  cout << "   appel compte : il y a              " << ctr   << " objets\n" ;
}
main()
{
   void fct () ;
   cpte_obj::compte () ;     // appel de la fonction membre statique compte
                             // alors qu'aucun objet de sa classe n'existe
   cpte_obj a ;
   cpte_obj::compte () ;
   fct () ;
   cpte_obj::compte () ;
   cpte_obj b ;
   cpte_obj::compte () ;
}
void fct()
{
   cpte_obj u, v ;
}
```

```
   appel compte : il y a              0 objets
++ construction : il y a maintenant   1 objets
   appel compte : il y a              1 objets
++ construction : il y a maintenant   2 objets
++ construction : il y a maintenant   3 objets
-- destruction  : il reste maintenant 2 objets
-- destruction  : il reste maintenant 1 objets
   appel compte : il y a              1 objets
++ construction : il y a maintenant   2 objets
```

```
   appel compte : il y a              2 objets
-- destruction  : il reste maintenant 1 objets
-- destruction  : il reste maintenant 0 objets
```

Définition et utilisation d'une fonction membre statique

9. LES FONCTIONS MEMBRE CONSTANTES

En langage C, le qualificatif const peut servir à désigner une variable dont on souhaite que la valeur n'évolue pas. Le compilateur est ainsi en mesure de rejeter d'éventuelles tentatives de modification de cette variable. Par exemple, avec cette déclaration :

```
const int n=20 ;
```

l'instruction suivante sera incorrecte :

```
n = 12 ;     // incorrecte
```

C++ généralise ce concept aux classes, ce qui signifie qu'on peut définir des "objets constants". Encore faut-il comprendre ce que l'on entend par là. En effet, dans le cas d'une variable ordinaire, il est relativement facile au compilateur d'identifier les opérations interdites (celles qui peuvent en modifier la valeur). Dans le cas d'un objet, par contre, les choses sont moins faciles, dans la mesure où les opérations sont généralement réalisées par les fonctions membre. Cela signifie donc que l'utilisateur doit préciser, parmi ces fonctions membre, lesquelles sont autorisées à opérer sur des objets constants. Il le fera en utilisant le mot const dans leur déclaration, comme dans cet exemple de définition d'une classe point :

```
class point
{     int x, y ;
  public :
      point (...) ;
      void affiche () const ;
      void deplace (...) ;
      ...
} ;
```

Ici, nous avons spécifié que la fonction affiche était utilisable pour un "point constant" ; en revanche, la fonction deplace, qui n'a pas fait l'objet d'une déclaration const ne le sera pas. Ainsi, avec ces déclarations :

```
point a ;
const point c ;
```

les instructions suivantes seront correctes :

```
a.affiche () ;
c.affiche () ;
a.deplace (...) ;
```

Celle-ci, en revanche, sera rejetée par le compilateur :

```
c.deplace (...) ;       // incorrecte
```

Remarques :

1) Le mécanisme que nous venons d'exposer s'applique aux fonctions membre volatiles et aux objets volatiles (mot clé volatile). Il suffit de transposer tout ce qui vient d'être dit en remplaçant le mot clé const par le mot clé volatile.

2) Il est possible de surdéfinir une fonction membre en se basant sur la présence ou l'absence du qualificatif const. Ainsi, dans notre classe point précédente, nous pouvons définir ces deux fonctions :

```
void affiche () const ;       // affiche I
void affiche () ;             // affiche II
```

Avec ces déclarations :

```
point a ;
const point c ;
```

l'instruction a.affiche () appellera la fonction II tandis que c.affiche () appellera la fonction I.

On notera bien que si seule la fonction void affiche() est définie, elle ne pourra en aucun cas être appliquée à un objet constant ; une instruction telle que c.affiche() serait alors rejetée en compilation. En revanche, si seule la fonction const void affiche() est définie, elle pourra être appliquée indifféremment à des objets constants ou non constants. Une telle démarche est manifestement logique ; en effet, elle peut se résumer ainsi :

- on ne court aucun risque en traitant un objet non constant comme s'il était constant,
- en revanche, il serait dangereux de faire à un objet constant ce qu'on a prévu de faire à un objet non constant.

10. LES POINTEURS SUR DES FONCTIONS MEMBRE

N.B. Ce paragraphe traite d'aspects relativement secondaires et peut, éventuellement, être ignoré dans un premier temps.

Le langage C permet de définir des pointeurs sur des fonctions. Leur emploi permet alors, en particulier, de programmer ce que l'on pourrait nommer des "appels variables" de fonctions. A titre de rappel, considérez ces déclarations :

```
int f1 (char, double) ;
int f2 (char, double) ;
   ...
int (* adf) (char, double) ;
```

La dernière signifie que adf est un pointeur sur une fonction recevant deux arguments, l'un de type char, l'autre de type double et fournissant un résultat de type int. Les affectations suivantes sont alors possibles :

```
adf = f1 ;        // affecte à adf l'adresse de la fonction f1
                  // on peut aussi écrire : adf = & f1 ;
adf = f2 ;        // affecte à adf l'adresse de la fonction f2
```

L'instruction :

```
(* adf) ('c', 5.25) ;
```

réalise l'appel de la fonction dont l'adresse figure dans adf, en lui fournissant en argument les valeurs 'c' et 5.25.

On notera que, comme les autres pointeurs du C, les pointeurs sur les fonctions sont "fortement typés", en ce sens que leur type précise à la fois la nature de la valeur de retour de la fonction et la nature de chacun de ses arguments.

Bien entendu, C++ vous offre toutes ces possibilités. Mais, de surcroît, il permet de les étendre au cas des fonctions membre. Cette extension passe néanmoins par une généralisation de la syntaxe précédente. En effet, il faut pouvoir tenir compte de ce qu'une fonction membre se définit :

- d'une part, comme une fonction ordinaire, c'est-à-dire d'après le type de ses arguments et de sa valeur de retour,
- d'autre part, d'après le type de la classe auquelle elle s'applique ; le type de l'objet l'ayant appelé constituant, en quelque sorte, le type d'un argument supplémentaire.

Ainsi, si une classe point comporte deux fonctions membre de prototypes :

```
void dep_hor (int) ;
void dep_vert (int) ;
```

la déclaration :

```
void (point::* adf) (int) ;
```

précisera que adf est un pointeur sur une fonction membre de la classe point recevant un argument de type int et ne renvoyant aucune valeur. Les affectations suivantes seront alors possibles :

```
adf = point::dep_hor ;  // ou adf = & point::dep_hor ;
adf = point::dep_vert ;
```

Enfin, si a est un objet de type point, une instruction telle que :

```
(a.*adf) (3) ;
```

provoquera, pour le point a, l'appel de la fonction membre dont l'adresse est contenue dans adf, en lui transmettant en argument la valeur 3.

EXERCICES

N.B. Les exercices marqués (C) sont corrigés en fin de volume.

1 - (C) Ecrivez une classe vecteur comportant :

- trois composantes de type double (privées),
- une fonction affiche,
- deux constructeurs :
  * l'un, sans arguments, initialisant chaque composante à 0,
  * l'autre, avec 3 arguments, représentant les composantes,

a) avec des fonctions membre indépendantes,

b) avec des fonctions membre "en ligne".

2 - (C) Ajoutez, à la première classe vecteur précédente, une fonction membre nommée prod_scal fournissant en résultat le produit scalaire de deux vecteurs.

3 - (C) Ajoutez, à la classe vecteur précédente (exercice 2), une fonction membre nommée somme permettant de calculer la somme de deux vecteurs.

4 - (C) Modifiez la classe vecteur précédente (exercice 3), de manière que toutes les transmissions de valeurs de type vecteur aient lieu :

a) par adresse,

b) par référence.

VII. CONSTRUCTION, DESTRUCTION, RECOPIE ET INITIALISATION DES OBJETS

En langage C, une variable peut être créée de deux façons :

- par une déclaration : elle est alors de classe **automatique** ou **statique** ; sa durée de vie est parfaitement définie par la nature et l'emplacement de sa déclaration,
- en faisant appel à des fonctions de gestion dynamique de la mémoire (malloc, calloc, free...) ; elle est alors dite **dynamique** ; sa durée de vie est contrôlée par le programme.

En langage C++, on retrouvera ces trois classes à la fois pour les variables ordinaires et pour les objets avec cette différence que la gestion dynamique fera appel aux opérateurs new et delete.

Ce sont ces différentes possibilités de création (donc aussi de destruction) des objets que nous allons étudier dans ce chapitre. Nous commencerons par examiner la création et la destruction des objets automatiques et statiques définis par une déclaration. Ceci nous amènera à préciser certains éléments déjà introduits dans les chapitres précédents (pour les objets automatiques). Puis nous montrerons comment créer et utiliser des objets dynamiques d'une manière comparable à celle employée pour créer des variables dynamiques ordinaires, mais en faisant appel à une syntaxe élargie de l'opérateur new.

Nous aborderons ensuite la notion de constructeur de recopie, lequel intervient dans les situations dites d'"initialisation d'un objet", c'est-à-dire lorsqu'il est nécessaire de réaliser une copie d'un objet existant. Nous verrons qu'il existe trois situations de ce type : transmission de la valeur d'un objet en argument d'une fonction, transmision de la valeur d'un objet en résultat d'une fonction, initialisation d'un objet lors de sa déclaration par un objet de même type ; la dernière possibilité n'étant qu'un cas particulier d'initialisation d'un objet au moment de sa déclaration.

Puis nous examinerons le cas des "objets membre", c'est-à-dire le cas où un type classe possède des membres donnée qui sont eux-même d'un type classe. Nous aborderons rapidement le cas des tableaux d'objets, notion d'autant moins importante qu'un tel tableau n'est pas lui-même un objet.

Enfin, nous fournirons quelques indications concernant les objets dits temporaires, c'est-à-dire des objets qui peuvent être créés au fil du déroulement du programme[1], sans que le programmeur l'ait explicitement demandé.

---

1 - En C, il existe déjà des variables temporaires, mais leur existence a moins d'importance que celle, en C++, des objets temporaires.

1. LES OBJETS AUTOMATIQUES ET STATIQUES

Examinons séparément :

- leur durée de vie, c'est-à-dire le moment où ils sont créés et celui où ils sont détruits,
- les éventuels appels des constructeurs et des destructeurs.

1.1 Leur durée de vie

Les règles s'appliquant aux variables ordinaires se transposent tout naturellement aux objets.

Les **objets automatiques** sont ceux créés par une déclaration :

- **dans une fonction** : c'était le cas dans nos exemples des précédents chapitres. L'objet est créé lors de la rencontre de sa déclaration, laquelle peut très bien, en C++, être située après d'autres instructions exécutables[2]. Il est détruit à la fin de l'exécution de la fonction.
- **dans un bloc** : de manière analogue au cas précédent, l'objet est créé lors de la rencontre de sa déclaration (là encore, celle-ci peut être précédée, au sein de ce bloc, d'autres instructions exécutables) ; il est détruit lors de la sortie du bloc.

Les **objets statiques** sont ceux créés par une déclaration située :

- en dehors de toute fonction,
- dans une fonction, mais assortie du qualificatif static.

Les objets statiques sont créés avant le début de l'exécution de la fonction main et ils sont détruits après la fin de son exécution.

1.2 Appel des constructeurs et des destructeurs

Rappelons que, si un objet possède un constructeur, sa déclaration (lorsque, comme nous le supposons pour l'instant, elle ne contient pas d'initialiseur) doit obligatoirement comporter les arguments correspondants. Par exemple, si une classe point comporte le constructeur de prototype :

```
point (int, int)
```

les déclarations suivantes seront incorrectes :

```
point a ;          // incorrect : le constructeur attend deux arguments
point b (3) ;      // incorrect (même raison)
```

Celle-ci, en revanche, conviendra :

```
point a(1, 7) ;    // correct car le constructeur possède deux arguments
```

S'il existe plusieurs constructeurs, il suffit que la déclaration comporte les arguments requis par l'un d'entre eux. Ainsi, si une classe point comporte les constructeurs suivants :

---

2 - La distinction entre instruction exécutable et instruction de déclaration n'étant pas toujours possible dans un langage comme C++ qui accepte, par exemple, une instruction telle que :

```
double * adr = new double [nelem = 2 * n+1] ;
```

```
point ( ) ;          // constructeur 1
point (int, int) ;   // constructeur 2
```

la déclaration suivante sera rejetée :

```
point a(5) ;         // incorrect : aucun constructeur à un argument
```

Mais celles-ci conviendront :

```
point a ;            // correct : appel du constructeur 1
point b(1, 7) ;      // correct : appel du constructeur 2
```

En ce qui concerne la "chronologie", on peut dire que :

- le **constructeur** est appelé **après la création** de l'objet,
- le **destructeur** est appelé **avant la destruction** de l'objet.

Remarque :

Une déclaration telle que :

```
point a ;         // attention, point a () est rejeté
```

est acceptable dans deux situations fort différentes :

- il n'existe pas de constructeur de point,
- il existe un constructeur de point sans argument.

1.3 Exemple

Voici un exemple de programme mettant en évidence la création et la destruction d'objets statiques et automatiques. Pour ce faire, nous y avons défini une classe nommée point, dans laquelle le constructeur et le destructeur affichent un message permettant de repérer :

- le moment de leur appel,
- l'objet concerné (nous avons fait en sorte que chaque objet de type point possède des valeurs différentes).

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
   point (int abs, int ord)          // constructeur ("inline")
      { x = abs ; y = ord ;
        cout << "++ Construction d'un point : " << x << " " << y << "\n" ;
      }

   ~point ()                         // destructeur ("inline")
      { cout << "-- Destruction du point    : " << x << " " << y << "\n" ;
```

```
      }
} ;

point a(1,1) ;                        // un objet statique de classe point
main()
{
   cout << "****** Début main *****\n" ;
   point b(10,10) ;                   // un objet automatique de classe point
   int i ;
   for (i=1 ; i<=3 ; i++)
      {  cout << "** Boucle tour numéro " << i << "\n" ;
         point b(i,2*i) ;             // objets créés dans un bloc
      }
   cout << "****** Fin main ******\n" ;
}
```

```
++ Construction d'un point : 1 1
****** Début main *****
++ Construction d'un point : 10 10
** Boucle tour numéro 1
++ Construction d'un point : 1 2
-- Destruction du point    : 1 2
** Boucle tour numéro 2
++ Construction d'un point : 2 4
-- Destruction du point    : 2 4
** Boucle tour numéro 3
++ Construction d'un point : 3 6
-- Destruction du point    : 3 6
****** Fin main ******
-- Destruction du point    : 10 10
-- Destruction du point    : 1 1
```

Construction et destruction d'objets statiques et automatiques

Remarque :

L'existence de constructeurs et de destructeurs conduit à des traitements qui n'apparaissent pas explicitement dans les instructions du programme. Par exemple, ici, une banale déclaration telle que :

```
point b(10, 10) ;
```

entraîne l'affichage d'un message.

Qui plus est, un "certain nombre de choses" se déroulent avant le début ou après l'exécution de la fonction main[3]. On pourrait, à la limite, concevoir une fonction main ne comportant que des déclarations (ce qui serait le cas de notre exemple, si nous supprimions l'instruction d'affichage du "tour de boucle"), et réalisant, malgré tout, un certain traitement.

3 - En toute rigueur, il en va déjà de même dans le cas d'un programme C (ouverture ou fermeture de fichiers par exemple) mais il ne s'agit pas alors de tâches programmées explicitement par l'auteur du programme ; dans le cas de C++, il s'agit de tâches programmées par le concepteur de la classe concernée.

2. LES OBJETS DYNAMIQUES

Nous avons déjà vu comment créer, utiliser et détruire en C++ des variables dynamiques scalaires (ou des tableaux de telles variables). Bien entendu, ces possibilités vont se généraliser aux structures et aux objets. Nous commencerons par le cas des structures, ce qui nous amènera en fait à effectuer un certain nombre de rappels sur l'utilisation des structures dynamiques en C.

2.1 Les structures dynamiques

Supposez que nous ayons défini la structure suivante :

```
struct chose
{       int x ;
        double y ;
        int t [5] ;
}
```

et que adr soit un pointeur sur des éléments de ce type, c'est-à-dire déclaré, en C, par :

```
struct chose * adr ;
```

ou plus simplement, en C++, par :

```
chose * adr ;
```

L'instruction :

```
adr = new chose ;
```

réalise une allocation dynamique d'espace mémoire pour un élément de type chose et affecte son adresse au pointeur adr.

L'accès aux différents champs de cette structure se fait à l'aide de l'opérateur ->. Ainsi, adr -> y en désignera le second champ. Rappelons que cette notation est en fait équivalente à (*adr). y.

L'espace mémoire ainsi alloué pourra être libéré par :

```
delete adr ;
```

2.2 Les objets dynamiques

Voyons tout d'abord ce qu'il y a de commun entre la création dynamique d'objets et celle de structures avant d'étudier les nouvelles possibilités de l'opérateur new.

a) Points communs avec les structures dynamiques

Le mécanisme que nous venons d'évoquer s'applique aux objets (au sens large), lorsqu'ils ne possèdent pas de constructeur. Ainsi, si nous définissons le type point suivant :

```
class point
{     int x, y ;
```

```
    public :
        void initialise (int, int) ;
        void deplace (int, int) ;
        void affiche ( ) ;
} ;
```

et si nous déclarons :

```
point * adr ;
```

nous pourrons créer dynamiquement un emplacement de type point (qui contiendra donc ici la place pour deux entiers) et affecter son adresse à adr par :

```
adr = new point ;
```

L'accès aux fonctions membre de l'objet pointé par adr se fera par des appels de la forme :

```
adr -> initialise (1, 3) ;
adr -> affiche ( ) ;
```

ou, éventuellement, sans utiliser l'opérateur -> , par :

```
(* adr).initialise (1, 3) ;
(* adr).affiche ( ) ;
```

Si l'objet contenait des membres donnée publics, on y accéderait de façon comparable.

Quant à la suppression de l'objet en question, elle se fera, ici encore, par :

```
delete adr ;
```

b) Les nouvelles possibilités des opérateurs new et delete

Nous avons déjà vu que la philosophie de C++ consiste à faire du constructeur (dès lors qu'il existe) un passage obligé lors de la création d'un objet. Il en va de même pour le destructeur lors de la destruction d'un objet.

Cette philosophie s'applique également aux objets dynamiques. Plus précisément :

- après l'allocation dynamique de l'emplacement mémoire requis, l'**opérateur new, appellera un constructeur de l'objet** ; ce constructeur sera déterminé par la nature des arguments qui figurent à la suite de son appel comme dans :

  ```
  new point (2, 5) ;
  ```

  On peut dire que le constructeur appelé est le même que celui qui aurait été appelé par une déclaration telle que :

  ```
  a = point (2, 5) ;
  ```

  Bien entendu, s'il n'existe pas de constructeur, ou s'il existe un constructeur sans argument, la syntaxe :

  ```
  new point     //  ou   new point ()
  ```

  sera acceptée. En revanche, si tous les constructeurs possèdent au moins un argument, cette syntaxe sera rejetée (elle ne sera acceptée que dans le cas où aucun constructeur n'existe).

  On retrouve là, en définitive, les mêmes règles que celles s'appliquant à la déclaration d'un objet.

- Avant la libération de l'emplacement mémoire correspondant, l'**opérateur delete appellera le destructeur.**

c) Exemple

Voici un exemple de programme qui crée dynamiquement un objet de type point dans la fonction main et qui le détruit dans une fonction fct (appelée par main). Les messages affichés permettent de mettre en évidence les moments auxquels sont appelés le constructeur et le destructeur.

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
   point (int abs, int ord)          // constructeur
      { x=abs ; y=ord ;
        cout << "++ Appel Constructeur \n" ;
      }
   ~point ()                         // destructeur (en fait, inutile ici)
      { cout << "-- Appel Destructeur \n" ;
      }
} ;
main()
{  void fct (point *) ;              // prototype fonction fct
   point * adr ;
   cout << "** Début main \n" ;
   adr = new point (3,7) ;           // création dynamique d'un objet
   fct (adr) ;
   cout << "** Fin main \n" ;
}
void fct (point * adp)
{  cout << "** Début fct \n" ;
   delete adp ;                      // destruction de cet objet
   cout << "** Fin fct \n" ;
}
```

```
** Début main
++ Appel Constructeur
** Début fct
-- Appel Destructeur
** Fin fct
** Fin main
```

Exemple de création dynamique d'objets

3. LE CONSTRUCTEUR DE RECOPIE

3.1 Présentation du constructeur de recopie

Nous avons vu comment C++ garantit l'appel d'un constructeur pour un objet créé par une déclaration ou par new. Ce point est fondamental puisqu'il donne la certitude qu'un objet ne pourra être créé, sans avoir été placé dans un "état initial convenable" (du moins jugé comme tel par le concepteur de l'objet).

Mais il existe des circonstances dans lesquelles il est nécessaire de construire un objet, même si le programmeur n'a pas prévu de constructeur pour cela. La situation la plus fréquente est celle où la valeur d'un objet doit être transmise en argument à une fonction. Dans ce cas, il est nécessaire de créer, dans un emplacement local à la fonction, un objet qui soit une copie de l'argument effectif. Le même problème se pose dans le cas d'un objet renvoyé par valeur comme résultat d'une fonction ; il faut alors créer, dans un emplacement local à la fonction appelante, un objet qui soit une copie du résultat. Nous verrons qu'il existe une troisième situation de ce type, à savoir le cas où un objet est initialisé, lors de sa déclaration, avec un autre objet de même type.

D'une manière générale, on regroupe ces trois situations sous le nom d'**initialisation par recopie**[4]. Une initialisation par recopie d'un objet étant donc la création d'un objet par recopie d'un objet existant de même type.

Pour réaliser une telle initialisation par recopie, C++ a prévu d'utiliser un constructeur particulier dit **constructeur de recopie**[5] (nous verrons plus loin la forme exacte qu'il doit posséder). Mais, si un tel constructeur n'existe pas, un traitement par défaut est prévu ; on peut dire qu'on utilise un constructeur de recopie par défaut.

En définitive, on peut dire que dans toute situation d'initialisation par recopie il y toujours appel d'un constructeur de recopie, mais il faut distinguer deux cas.

a) Il n'existe pas de constructeur approprié

Il y appel d'un **constructeur de recopie par défaut**, généré automatiquement par le compilateur. Ce constructeur se contente d'effectuer une copie de chacun des membres, laquelle est analogue, en définitive, à celle qui est mise en place (par défaut) lors d'une affectation entre objets de même type. Naturellement cela posera les mêmes problèmes dans le cas d'objets contenant des pointeurs sur des emplacements alloués dynamiquement : on aura simplement affaire à une "copie superficielle", c'est-à-dire que seules les valeurs des pointeurs seront recopiées, les emplacements pointés ne le seront pas ; ils risquent alors, par exemple, d'être détruits deux fois.

b) Il existe un constructeur approprié

Il existe donc un **constructeur de recopie** que vous aurez **fourni explicitement** dans votre classe. Il doit s'agir d'un constructeur ayant un seul argument[6] du type de cette classe ; de plus, il est indispensable que cet argument soit transmis par référence, ce qui signifie que l'en-tête du constructeur de recopie doit être obligatoirement de l'une de ces deux formes (si la classe concernée se nomme point) :

```
point (point & )          point (const point & )
```

---

4 - Nous aurions pu nous limiter au terme "initialisation" s'il n'existait pas des situations où l'on peut initialiser un objet avec une valeur ou un objet d'un type différent...

5 - En anglais "copy constructor".

6 - En toute rigueur, la norme ANSI accepte également un constructeur disposant d'arguments supplémentaires, pourvu que ces derniers possèdent des valeurs par défaut.

Dans ce cas, ce constructeur est appelé de manière habituelle, après la création de l'objet. Bien entendu, aucune recopie n'est faite de façon automatique, pas même une recopie superficielle, contrairement à la situation précédente : c'est à ce constructeur de prendre en charge l'intégralité du travail (copie superficielle et copie profonde).

Remarques :

1) Notez bien que C++ impose au constructeur en question que son unique argument soit transmis par référence (ce qui est logique puisque, sinon, l'appel du constructeur de recopie impliquerait une initialisation par recopie de l'argument, donc, en définitive, un appel du constructeur de recopie qui, lui-même, etc.)

   Quoi qu'il en soit, la forme suivante serait rejetée en compilation :

```
point (point) ;    // incorrect
```

2) Les deux formes précédentes (point (point &) et point (const point &)) pourraient exister au sein d'une même classe. Dans ce cas, la première serait utilisée en cas d'initialisation d'un objet par un objet quelconque, tandis que la seconde serait utilisée en cas d'initialisation par un objet constant. En général, compte tenu de ce que, logiquement, un tel constructeur de recopie n'a aucune raison de vouloir modifier l'objet reçu en argument, il est conseillé de ne définir que la seconde forme ; dans ce cas, en effet, elle restera applicable aux deux situations évoquées (une fonction prévue pour un objet constant peut toujours s'appliquer à un objet variable – la réciproque étant naturellement fausse).

3) Nous avons déjà rencontré des situations de recopie dans le cas de l'affectation. Mais, alors, les deux objets concernés existaient déjà ; l'affectation n'est donc pas une situation d'initialisation par recopie, telle que nous venons de la définir. Bien que les deux opérations possèdent un traitement par défaut semblable (copie superficielle), la prise en compte d'une copie profonde passe par des mécanismes différents : définition d'un constructeur de recopie pour l'initialisation, surdéfinition de l'opérateur = pour l'affectation (ce que nous apprendrons à faire dans le chapitre consacré à la surdéfinition des opérateurs).

3.2 Premier exemple d'utilisation du constructeur de recopie : objet transmis par valeur

Nous vous proposons de comparer les deux situations que nous venons d'évoquer : constructeur de recopie par défaut, constructeur de recopie défini dans la classe. Pour ce faire, nous allons utiliser une classe vect permettant de gérer des tableaux d'entiers de taille "variable" (on devrait plutôt dire de taille définissable lors de l'exécution car, une fois définie, cette taille ne changera plus). Nous souhaitons que l'utilisateur de cette classe déclare un tableau sous la forme :

```
vect t (dim) ;
```

où dim est une expression entière représentant sa taille.

Il paraît alors naturel de prévoir pour vect :

- en membres donnée, la taille du tableau et un pointeur sur ses éléments, lesquels verront leurs emplacements alloués dynamiquement,
- un constructeur recevant un argument entier chargé de cette allocation dynamique,
- un destructeur libérant l'emplacement alloué par le constructeur.

Cela nous conduit à une "première ébauche" :

```
class vect
{     int nelem ;
      double * adr ;
  public :
      vect (int n) ;
      ~vect ( ) ;
} ;
```

a) Emploi du constructeur de recopie par défaut

Voici un exemple d'utilisation d'une telle classe (nous avons ajouté des affichages de messages pour suivre à la trace les constructions et destructions d'objets). Ici, nous nous contentons de transmettre par valeur un objet de type vect à une fonction ordinaire nommée fct, laquelle ne fait rien d'autre que d'afficher un message indiquant son appel.

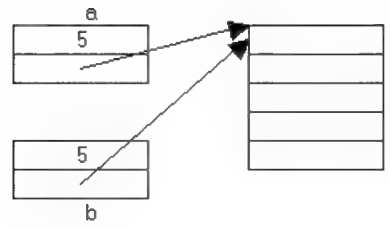
```
#include <iostream.h>
class vect
{
   int nelem ;                        // nombre d'éléments
   double * adr ;                     // pointeur sur ces éléments
  public :
   vect (int n)                       // constructeur "usuel"
   { adr = new double [nelem = n] ;
      cout << "+ const. usuel - adr objet : " << this
           << " - adr vecteur : " << adr << "\n" ;
   }
   ~vect ()                           // destructeur
    { cout << "- Destr. objet - adr objet : "
           << this << " - adr vecteur : " << adr << "\n" ;
      delete adr ;
    }
} ;
void fct (vect b)
{ cout << "*** appel de fct ***\n" ;
}
main()
{ vect a(5) ;
  fct (a) ;
}
```

```
+ const. usuel - adr objet : 0x40e20ffa - adr vecteur : 0x42940004
*** appel de fct ***
- Destr. objet - adr objet : 0x40e20ff4 - adr vecteur : 0x42940004
- Destr. objet - adr objet : 0x40e20ffa - adr vecteur : 0x42940004
```

Comme vous pouvez le constater, l'appel :

```
fct (a) ;
```

a créé un nouvel objet, dans lequel on a recopié les valeurs des membres nelem et adr de a. La situation peut être schématisée ainsi (nous désignons par b le nouvel objet ainsi créé) :



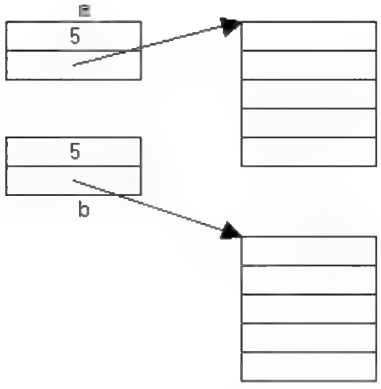
A la fin de l'exécution de la fonction main, il y a appel du destructeur ~ point, d'abord pour a, ce qui libère l'emplacement pointé par adr, puis pour b, ce qui libère... le même emplacement. Cette tentative constitue une erreur (d'exécution) dont les conséquences varient avec l'implémentation.

b) Définition d'un constructeur de recopie

Nous pouvons éviter le problème évoqué en faisant en sorte que l'appel :

```
fct (a) ;
```

conduise à créer "intégralement" un nouvel objet de type vect, avec non seulement ses membres donnée nelem et adr, mais également son propre emplacement de stockage des valeurs du tableau. Autrement dit, nous souhaitons aboutir à cette situation :



Pour ce faire, nous définissons, au sein de la classe vect, un constructeur par recopie, de la forme :

```
vect (const vect &) ;       // ou, a la rigueur    vect (vect &)
```

dont nous savons qu'il sera appelé dans toute situation d'initialisation donc, en particulier, lors de l'appel de fct.

Ce constructeur (appelé après la création d'un nouvel objet[7]) doit :

- créer dynamiquement un nouvel emplacement dans lequel il recopie les valeurs correspondant à l'objet reçu en argument,
- renseigner convenablement les membres donnée du nouvel objet (nelem = valeur du membre nelem de l'objet reçu en argument, adr = adresse du nouvel emplacement).

Introduisons ce constructeur de recopie dans notre précédent exemple :

```
#include <iostream.h>
class vect
{
   int nelem ;                      // nombre d'éléments
   double * adr ;                   // pointeur sur ces éléments
  public :
   vect (int n)                     // constructeur "usuel"
   { adr = new double [nelem = n] ;
     cout << "+ const. usuel   - adr objet : " << this
          << " - adr vecteur : " << adr << "\n" ;
   }
   vect (const vect & v)            // constructeur de recopie
   { adr = new double [nelem = v.nelem] ;                // création nouvel
objet
     int i ; for (i=0 ; i<nelem ; i++) adr[i]=v.adr[i] ; // recopie de l'ancien
     cout << "+ const. recopie - adr objet : " << this
          << " - adr vecteur : " << adr << "\n" ;
   }
   ~vect ()                         // destructeur
   { cout << "- Destr. objet   - adr objet : "
          << this << " - adr vecteur : " << adr << "\n" ;
     delete adr ;
   }
} ;
void fct (vect b)
{ cout << "*** appel de fct ***\n" ; }
main()
{ vect a(5) ;   fct (a) ;
}
```

```
+ const. usuel   - adr objet : 0x44ec0ffa - adr vecteur : 0x469e0004
+ const. recopie - adr objet : 0x44ec0ff4 - adr vecteur : 0x46a10004
*** appel de fct ***
- Destr. objet   - adr objet : 0x44ec0ff4 - adr vecteur : 0x46a10004
- Destr. objet   - adr objet : 0x44ec0ffa - adr vecteur : 0x469e0004
```

7 - Notez bien que le constructeur n'a pas à créer l'objet lui-même, c'est-à-dire ici les membres int et adr, mais simplement les parties soumises à la gestion dynamique.

Vous constatez que, cette fois, chaque objet possédant son propre emplacement mémoire, les destructions successives se déroulent sans problème.

Remarque

Si nous avons réglé le problème de l'initialisation d'un objet de type vect par un autre objet du même type, nous n'avons pas pour autant réglé celui qui se poserait en cas d'affectation entre objets de type vect. Comme nous l'avons déjà signalé à plusieurs reprises, ce dernier point ne peut se résoudre que par la surdéfinition de l'opérateur =.

3.3 Deuxième exemple d'utilisation du constructeur de recopie : objet transmis en valeur de retour d'une fonction

Lorsque la transmission d'un argument ou d'une valeur de retour d'une fonction a lieu par valeur, elle met en œuvre une recopie. Lorsque celle-ci concerne un objet, cette recopie est, comme nous l'avons dit, réalisée soit par le constructeur de recopie par défaut, soit, le cas échéant, par le constructeur de recopie prévu pour l'objet.

Dans le cas où un objet comporte une partie dynamique, l'emploi de la recopie par défaut conduit, comme nous l'avons déjà vu, à une "copie superficielle" ne recopiant que les membres de l'objet. Les risques de double libération d'un emplacement mémoire sont alors les mêmes que ceux évoqués dans le paragraphe 3.2. Mais, de surcroît, pour la partie dynamique de l'objet, on perd ici le bénéfice de la protection contre des modifications qu'offre la transmission par valeur. En effet, dans ce cas, la fonction concernée reçoit bien une copie de l'adresse de l'emplacement mais, par le biais de ce pointeur, elle peut tout à fait modifier le contenu de l'emplacement lui-même (revoyez, par exemple, le schéma du paragraphe 3.2a, dans lequel a jouait le rôle d'un argument et b celui de sa recopie).

Voici un exemple de programme faisant appel à une classe point, dotée d'une fonction membre nommée symetrique, fournissant en retour un point symétrique de celui l'ayant appelé. Notez bien qu'ici, contrairement à l'exemple précédent, le constructeur de recopie n'est pas indispensable au bon fonctionnement de notre classe (qui ne comporte aucune partie dynamique) : il ne sert qu'à illustrer le mécanisme de son appel.

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)      // constructeur "usuel"
     { x=abs ; y=ord ;
       cout << "++ Appel Const. usuel   " << this << " " << x << " " << y <<
"\n" ;
     }
   point (point & p)                 // constructeur de recopie
     { x=p.x ; y=p.y ;
       cout << "++ Appel Const. recopie " << this << " " << x << " " << y <<
"\n" ;
```

```
      }
   ~point ()
     { cout << "-- Appel Destr.          " << this << " " << x << " " << y <<
"\n" ;
      }
   point symetrique () ;
} ;
point point::symetrique ()
{ point res ; res.x = -x ; res.y = -y ; return res ;
}

main()
{   point a(1,3), b ;
    cout << "** avant appel de symetrique\n" ;
    b = a.symetrique () ;
    cout << "** après appel de symetrique\n" ;
}
```

```
++ Appel Const. usuel   0x41000ffc 1 3
++ Appel Const. usuel   0x41000ff8 0 0
** avant appel de symetrique
++ Appel Const. usuel   0x41000fe2 0 0
++ Appel Const. recopie 0x41000ff4 -1 -3
-- Appel Destr.         0x41000fe2 -1 -3
** après appel de symetrique
-- Appel Destr.         0x41000ff4 -1 -3
-- Appel Destr.         0x41000ff8 -1 -3
-- Appel Destr.         0x41000ffc 1 3
```

Appel du constructeur de recopie en cas de transmission par valeur

4. INITIALISATION D'UN OBJET LORS DE SA DÉCLARATION

N.B. Ce paragraphe peut être ignoré dans un premier temps.

En langage C, on peut initialiser une variable au moment de sa déclaration comme dans :

```
int n = 12 ;
```

En théorie, C++ permet de faire de même avec les objets, en ajoutant un initialiseur lors de leur déclaration. Mais si le rôle d'un tel initialiseur va de soi dans le cas de variables classiques (il ne s'agit que d'en fournir la ou les valeurs), il n'en va plus de même dans le cas d'un objet ; en effet, il ne s'agit plus de se contenter d'initialiser simplement ses membres mais plutôt de fournir, sous une forme peu naturelle, des arguments pour un constructeur. De plus, C++ n'impose aucune restriction sur le type de l'initialiseur qui pourra donc, éventuellement, être du même type que l'objet initialisé : le constructeur utilisé sera alors le constructeur de recopie présenté précédemment.

Voyons cela plus en détail. Considérons d'abord cette classe (munie d'un constructeur usuel) :

```
class point
{     int x, y ;
  public :
```

```
        point (int abs) { x = abs ; y = 0 ; }
        .....
} ;
```

Nous avons déjà vu quel serait le rôle d'une déclaration telle que :

```
point a(3) ;
```

C++ nous autorise également à écrire :

```
point a = 3 ;
```

Cette déclaration entraîne :

- la création d'un objet nommé a,
- l'appel du constructeur auquel on transmet en argument la valeur de l'initialiseur, ici 3.

En définitive, les deux déclarations :

```
point a(3) ;
point a = 3 ;
```

sont équivalentes.

D'une manière générale, lorsque l'on déclare un objet avec un **initialiseur**, ce dernier peut être une **expression** d'un **type quelconque**, à condition qu'il existe un constructeur à un seul argument de ce type.

Cela s'applique donc également à une situation telle que :

```
point a ;
point b = a ;      // on initialise b avec l'objet a de même type
```

Manifestement, on aurait obtenu le même résultat en déclarant :

```
point b(a) ;       // on crée l'objet b, en utilisant le constructeur par
recopie
                   //  de la classe point, auquel on transmet l'objet a
```

Quoi qu'il en soit, ces deux déclarations (point b= a et point b(a)) entraînent effectivement la création d'un objet de type point, suivie de l'appel du constructeur par recopie de point (celui par défaut ou, le cas échéant, celui qu'on y a défini) auquel on transmet en argument l'objet a.

Remarques :

1) Il ne faut pas confondre l'initialiseur d'une classe avec celui employé en C pour donner des valeurs initiales à un tableau :

   ```
   int t[5] = {3, 5, 11, 2, 0} ;
   ```

   ou à une structure. Celui-ci est toujours utilisable en C++, y compris pour les structures comportant des fonctions membre. Il est même applicable à des classes ne disposant pas de constructeur et dans lesquelles tous les membres sont publics ; en pratique, cette possibilité ne présente guère d'intérêt.

2) Supposons qu'une classe point soit munie d'un constructeur à deux arguments entiers et considérons la déclaration :

   ```
   point a = point (1, 5) ;
   ```

Il s'agit bien d'une déclaration comportant un initialiseur constitué d'une expression de type point. On pourrait logiquement penser qu'elle entraîne l'appel d'un constructeur de recopie (par défaut ou effectif) en vue d'initialiser l'objet a nouvellement créé avec l'objet temporaire point (1,5).

En fait, dans ce cas précis d'initialisation d'un objet par appel explicite du constructeur, C++ a prévu de traiter cette déclaration comme :

```
point a(1, 5) ;
```

Autrement dit, il y a création d'un seul objet a et appel du constructeur ("usuel") pour cet objet. Aucun constructeur de recopie n'est appelé.

Cette démarche est, au demeurant, assez naturelle et simplificatrice. Elle n'en demeure pas moins une exception par opposition à celle qui est mise en œuvre dans :

```
point a = b ;
```

ou dans :

```
point a = b + point (1, 5)
```

lorsque nous aurons appris à donner un sens à une expression telle que b + point (1, 5) (elle suppose la "surdéfinition" de l'opérateur + pour la classe point).

5. OBJETS MEMBRE

5.1 Introduction

Il est tout à fait possible qu'une classe possède un membre donnée qui soit lui-même de type classe. Par exemple, ayant défini :

```
class point
{     int x, y ;
  public :
      int init (int, int) ;
      void affiche ( ) ;
} ;
```

nous pouvons définir :

```
class cercle
{     point centre ;
      int rayon ;
  public :
      void affrayon ( ) ;
      ...
} ;
```

Si nous déclarons alors :

```
cercle c ;
```

l'objet c possède un membre donnée privé centre, de type point. L'objet c peut accéder classiquement à la méthode affrayon par c.affrayon. En revanche, il ne poura pas accéder à la méthode init du membre centre

car centre est privé. Si centre était public, on pourrait accéder aux méthodes de centre par c.centre.init () ou c.centre.affiche ().

D'une manière générale, la situation d'objets membre correspond à une relation entre classe du type relation de possession (on dit aussi "relation a" – du verbe avoir). Effectivement, ici, on peut bien dire qu'un cercle possède (a) un centre (de type point). Ce type de relation s'oppose au type de relation qui sera induite par l'héritage, dont on dira qu'il s'agit de "relation est" (du verbe être).

Voyons maintenant comment sont mis en œuvre les constructeurs des différents objets lorsqu'ils existent.

5.2 Mise en œuvre des constructeurs et des destructeurs

Supposons, cette fois, que notre classe point ait été définie avec un constructeur :

```
class point
{     int x, y ;
  public :
      point (int, int) ;
} ;
```

Nous ne pouvons plus définir la classe cercle précédente sans constructeur. En effet, si nous le faisions, son membre centre se verrait certes attribuer un emplacement (lors d'une création d'un objet de type cercle) mais son constructeur ne pourrait être appelé (quelles valeurs pourrait-on lui transmettre ?).

Il faut donc :

- d'une part, définir un constructeur pour cercle,
- d'autre part, spécifier les arguments à fournir au constructeur de point : ceux-ci doivent être choisis obligatoirement parmi ceux fournis à cercle.

Voici ce que pourrait être la définition de cercle et de son constructeur :

```
class cercle
{     point centre ;
      int rayon ;
  public :
      cercle (int, int, int) ;
} ;
cercle::cercle (int abs, int ord, int ray) : centre (abs, ord)
{ ...
}
```

Vous voyez que l'en-tête de cercle spécifie, après les deux-points, la liste des arguments qui seront transmis à point.

Les constructeurs seront appelés dans l'ordre suivant : point, cercle. S'il existe des destructeurs, ils seront appelés dans l'ordre inverse.

Voici un exemple complet :

```
#include <iostream.h>
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0)
    { x=abs ; y=ord ;
      cout << "Constr. point " << x << " " << y << "\n" ;
    }
} ;

class cercle
{ point centre ;
  int rayon ;
 public :
  cercle (int , int , int) ;
} ;

cercle::cercle (int abs, int ord, int ray) : centre(abs, ord)
   { rayon = ray ;
     cout << "Constr. cercle " << rayon << "\n" ;
   }

main()
{  cercle a (1,3,9) ;
}
```

```
Constr. point 1 3
Constr. cercle 9
```

Appel des différents constructeurs dans le cas d'objets membre

Remarques :

1) Si point dispose d'un constructeur sans argument, le constructeur de cercle peut ne pas spécifier d'argument à destination du constructeur de centre qui sera appelé automatiquement.

2) Dans le cas d'objets comportant plusieurs objets membre, la sélection des arguments destinés aux différents constructeurs se fait en séparant chaque liste par une virgule. En voici un exemple :

```
class A                          class B
{     ...                  {     ...
      A (int) ;                        B (double, int) ;
      ...                        ...
} ;                          } ;

class C
{     A a1 ;
      B b ;
      A a2 ;
      ...
      C (int n, int p, double x, int q, int r) : a1(p), b(x,q), a2(r)
```

```
        { ..... }
       ...
} ;
```

Ici, nous avons, par souci de simplification de l'écriture, supposé que le constructeur de C était "en ligne". Parmi les arguments n, p, x, q et r qu'il reçoit, p sera transmis au constructeur A de a1, x et q au constructeur B de b puis r au constructeur A de a2. Notez bien que l'ordre dans lequel ces trois constructeurs sont exécutés n'est pas imposé par le langage (attention aux éventuels effets de bord !). Toutefois, le constructeur C, quant à lui, ne sera bien exécuté qu'après les trois autres (l'ordre des imbrications est toujours respecté).

3) La syntaxe que nous venons de décrire pour transmettre des arguments à un constructeur d'un objet membre peut en fait s'appliquer à n'importe quel membre même s'il ne s'agit pas d'un objet. Par exemple :

```
class point
{  int x, y ;
  public :
   point (int abs=0, int ord=0) : x(abs), y(ord) {}
   .....
} ;
```

L'appel du constructeur point provoquera l'initialisation des membres x et y avec respectivement les valeurs abs et ord. Son corps est vide ici puisqu'il n'y a rien de plus à faire pour remplacer notre constructeur classique :

```
point (int abs=0, int ord=0) { x=abs ; y=ord ; }
```

Cette possibilité peut devenir indispensable dans les cas suivants :

- initialisation d'un membre donnée constant,
- initialisation d'un membre donnée qui est une référence : en effet, on ne peut qu'initialiser une telle référence, jamais lui affecter une nouvelle valeur (revoyez éventuellement le paragraphe correspondant dans le chapitre IV).

5.3 Cas du constructeur de recopie

Nous avons vu que, pour toute classe, il est prévu un constructeur de recopie par défaut, lequel est appelé en l'absence de constructeur de recopie effectif. Son rôle est simple dans le cas d'objets ne comportant pas d'objets membre, puisqu'il s'agit alors de recopier les valeurs des différents membres donnée.

Lorsque l'objet comporte des objets membre, la recopie (par défaut) se fait membre par membre[8] ; autrement dit, si l'un des membres est lui-même un objet, on le recopiera en appelant **son propre constructeur de recopie** (qui pourra être soit un constructeur par défaut, soit un constructeur défini dans la classe correspondante).

Cela signifie que la construction par recopie (par défaut) d'un objet sera satisfaisante dès lors qu'il ne contient pas de pointeurs sur des parties dynamiques, même si certains de ses objets membre en comportent (à condition qu'ils soient, quant à eux, convenablement munis des constructeurs par recopie appropriés).

Si, en revanche, l'objet contient des pointeurs, il faudra le munir d'un constructeur de recopie approprié. Ce dernier devra alors prendre en charge l'intégralité de la recopie de l'objet. Cependant, on pourra pour cela,

8 - En anglais, on parle de "member-wise copy". Avant la version 2.0 de C++, la copie se faisait bit à bit ("bitwise copy"), ce qui n'était pas toujours satisfaisant.

le cas échéant, transmettre les informations nécessaire aux constructeurs par recopie (par défaut ou non) de certains de ses membres en utilisant la technique décrite dans le paragraphe 5.2.

6. LES TABLEAUX D'OBJETS

N.B. Ce paragraphe peut être ignoré dans un premier temps.

En C++, un tableau peut posséder des éléments de n'importe quel type, y compris de type classe, ce qui conduit alors à des tableaux d'objets. Ce concept ne présente pas de difficultés particulières au niveau des notations que nous allons nous contenter de rappeler sur un exemple. En revanche, il nous faudra préciser certains points relatifs à l'appel des constructeurs et aux initialiseurs.

6.1 Notations

Soit une classe point, sans constructeur, définie par :

```
class point
{     int x, y ;
  public :
      void init (int, int) ;
      void affiche ( ) ;
} ;
```

Nous pouvons déclarer un tableau courbe de vingt objets de type point par :

```
point courbe [20] ;
```

Si i est un entier, la notation courbe[i] désignera un objet de type point. L'instruction :

```
courbe[i].affiche () ;
```

appellera le membre init pour le point courbe[i] (les priorités relatives des opérateurs . et [] permettent de s'affranchir de parenthèses). De même, on pourra "afficher" tous les points par :

```
for (i = 0 ; i < 20 ; i++) courbe[i].affiche() ;
```

Remarque :

Un tableau d'objets n'est pas un objet. Dans l'esprit de la P.O.O. pure, ce concept n'existe pas, puisqu'on ne manipule que des objets. En revanche, il reste toujours possible de définir une classe dont un des membres est un tableau d'objets. Ainsi, dans notre cas, nous pourrions définir un type courbe par :

```
class courbe
{   point p[20] ;
      ...
} ;
```

6.2 Constructeurs et initialiseurs

Nous venons de voir la signification de la déclaration :

```
point courbe[20] ;
```

dans le cas où point est une classe sans constructeur.

Si la classe comporte un constructeur sans argument, celui-ci sera appelé successivement pour chacun des éléments (de type point) du tableau courbe. En revanche, si aucun des constructeurs de point n'est un constructeur sans argument, la déclaration précédente conduira à une erreur de compilation. Ceci s'explique par le fait que, dans ce cas, C++ n'est plus en mesure de garantir le passage par un constructeur, dès lors que la classe concernée (point) en comporte au moins un.

Il est cependant possible de compléter une telle déclaration par un initialiseur comportant une liste de valeurs ; chaque valeur sera transmise à un constructeur approprié (les valeurs peuvent donc être de types quelconques, éventuellement différents les uns des autres, dans la mesure où il existe le constructeur correspondant). Pour les tableaux de classe automatique, les valeurs de l'initialiseur peuvent être une expression quelconque (pour peu qu'elle soit calculable au moment où l'on en a besoin). En outre, l'initialiseur peut comporter moins de valeurs que le tableau n'a d'éléments[9]. Dans ce cas, il y a appel du constructeur sans argument (qui doit donc exister) pour les éléments auxquels ne correspond aucune valeur.

Voici un exemple illustrant ces possibilités (ici, nous avons choisi un constructeur disposant d'arguments par défaut : il remplace trois constructeurs à zéro, un et deux arguments).

```
#include <iostream.h>
class point
{   int x, y ;
  public :
    point (int abs=0, int ord=0)      // constructeur (0, 1 ou 2 arguments)
      { x=abs ; y =ord ;
        cout << "++ Constr. point : " << x << " " << y << "\n" ;
      }
} ;
main()
{  int n = 3 ;
   point courbe[5] = { 7, n, 2*n+5 } ;
}
```

```
++ Constr. point : 7 0
++ Constr. point : 3 0
++ Constr. point : 11 0
++ Constr. point : 0 0
++ Constr. point : 0 0
-- Destr.  point : 0 0
-- Destr.  point : 0 0
-- Destr.  point : 11 0
-- Destr.  point : 3 0
-- Destr.  point : 7 0
```

[9] - Mais, pour l'instant, les éléments manquants doivent obligatoirement être les derniers.

Construction et initialisation d'un tableau d'objets (version 2.0)

6.3 Cas des tableaux dynamiques d'objets

Si l'on dispose d'une classe point, on peut créer dynamiquement un tableau de points en faisant appel à l'opérateur new. Par exemple :

```
point * adcourbe = new point[20] ;
```

alloue l'emplacement mémoire nécessaire à vingt objets (consécutifs) de type point et place l'adresse du premier de ces objets dans adcourbe.

Là encore, si la classe point comporte un constructeur sans argument, ce dernier sera appelé pour chacun des vingt objets. Si, en revanche, aucun des constructeurs de point n'est un constructeur sans argument, l'instruction précédente conduira à une erreur de compilation. Bien entendu, aucun problème particulier ne se posera si la classe point ne comporte aucun constructeur.

Par contre, il n'existe ici aucune possibilité de fournir un initialiseur, alors que, comme nous l'avons vu dans le paragraphe 6.2, ceci est possible dans le cas de tableaux automatiques ou statiques.

Pour détruire notre tableau d'objets, il suffira de l'instruction (notez la présence de [] qui précise que l'on a affaire à un tableau d'objets) :

```
delete [] adcourbe
```

Celle-ci provoquera l'appel du destructeur de point et la libération de l'espace correspondant **pour chacun des éléments du tableau.**

7. LES OBJETS TEMPORAIRES

N.B. Ce paragraphe peut être ignoré dans un premier temps.

Lorsqu'une classe dispose d'un constructeur, ce dernier peut être appelé explicitement (avec la liste d'arguments nécessaires). Dans ce cas, il y a alors création d'un objet temporaire. Par exemple, si nous supposons qu'une classe point possède le constructeur :

```
point (int, int) ;
```

nous pouvons, si a est un objet de type point, écrire une affectation telle que :

```
a = point (1, 2) ;
```

Dans une telle instruction, l'évaluation de l'expression :

```
point (1, 2)
```

conduit à :

- la création d'un objet temporaire de type point (il a une adresse précise, mais il n'est pas accessible au programme)[10],
- l'appel du constructeur point, pour cet objet temporaire, avec transmission des arguments spécifiés (ici 1 et 2),
- la recopie de cet objet temporaire dans a (affectation d'un objet à un autre de même type).

Quant à l'objet temporaire ainsi créé, il n'a plus d'intérêt dès que l'instruction d'affectation est exécutée. Il pourra donc être détruit à tout moment. Néanmoins, la définition du langage C++ ne spécifie pas à quel moment ! Ceci peut avoir une importance dans le cas où l'on tient compte du nombre d'objets d'une même classe existant à un moment donné, ou encore lorsque l'on emploie des "compteurs de références" dont nous parlons en annexe.

Voici un exemple de programme montrant l'emploi d'objets temporaires. Remarquez qu'ici, nous avons prévu, dans le constructeur et le destructeur de notre classe point, d'afficher non seulement les valeurs de l'objet mais également son adresse.

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
   point (int abs, int ord)          // constructeur ("inline")
      { x = abs ; y = ord ;
        cout << "++ Constr. point " << x << " " << y
             << " à l'adresse : " << this << "\n" ;
      }
   ~point ()                         // destructeur ("inline")
      { cout << "-- Destr.  point " << x << " " << y
             << " à l'adresse : " << this << "\n" ;
      }
} ;



main()
{
   point a(0,0) ;                    // un objet automatique de classe point
   a = point (1, 2) ;                // un objet temporaire
   a = point (3, 5) ;                // un autre objet temporaire
   cout << "****** Fin main ******\n" ;
}
```

```
++ Constr. point 0 0 à l'adresse : 0x40eb0ffc
++ Constr. point 1 2 à l'adresse : 0x40eb0ff8
++ Constr. point 3 5 à l'adresse : 0x40eb0ff4
****** Fin main ******
-- Destr.  point 3 5 à l'adresse : 0x40eb0ff4
```

10 - En fait, il en va de même lorsque l'on réalise une affectation telle que y = a * x + b. Il y a bien création d'un emplacement temporaire destiné à recueillir le résultat de l'évaluation de l'expression a * x + b.

```
-- Destr.  point 1 2 à l'adresse : 0x40eb0ff8
-- Destr.  point 3 5 à l'adresse : 0x40eb0ffc
```

Exemple de création d'objets temporaires

On y voit clairement que les deux affectations de la fonction main entraînent la création d'un objet temporaire, distinct de a. Ici, le compilateur utilisé n'a prévu de détruire les objets temporaires qu'à la fin de l'exécution de la fonction concernée (main), de sorte que les messages de destruction n'apparaissent qu'après celui de fin ; bien entendu, il pourrait en aller autrement avec d'autres compilateurs.

Remarques

1) Répétons que, dans une affectation telle que :

```
a = point (1, 2) ;
```

l'objet a existe déjà. Il n'a donc pas à être créé et il n'y a pas d'appel de constructeur à ce niveau pour a.

2) Les remarques faites à propos des risques que comporte une affectation entre objets, notamment dans le cas où ils comportent des parties dynamiques[11] restent valables ici. La même solution pourra y être apportée, à savoir la surdéfinition de l'opérateur d'affectation.

3) Il existe d'autres circonstances dans lesquelles sont créés des objets temporaires, à savoir :

- transmission de la valeur d'un objet en argument d'une fonction ; dans ce cas, il y a création d'un objet temporaire au sein de la fonction concernée ;
- transmission d'un objet en valeur de retour d'une fonction ; dans ce cas, il y a création d'un objet temporaire au sein de la fonction appelante.

Dans les deux cas, l'objet temporaire est initialisé par appel du constructeur de recopie.

4) La présence d'objets temporaires (dont le moment de destruction n'est pas parfaitement imposé par la norme) peut rendre difficile le dénombrement exact d'objets d'une classe donnée.

5) La norme ANSI autorise les compilateurs à supprimer certaines créations d'objets temporaires, notamment dans des situations telles que :

```
    f (point (1,2)) ;  // appel d'une fonction attendant un point
                       //  avec un argument qui est un objet temporaire
                       // on peut ne pas créer point(1,2) dans la fonction
appelante

    return (point (3, 5)) ;  // renvoi de la valeur d'un point
                             // on peut ne pas créer point(3,5) dans la
fonction
```

EXERCICES

N.B. Les exercices marqués (C) sont corrigés en fin de volume.

11 - Nous incluons dans ce cas les objets dont un membre (lui-même objet) comporte une partie dynamique.

1 - Comme le suggère la remarque du paragraphe 1.3, écrivez une fonction main qui, bien que ne contenant que des déclarations (voire une seule déclaration), n'en effectue pas moins un certain traitement (par exemple affichages).

2 - Expérimentez le programme du paragraphe 2 pour voir comment sont traités les objets temporaires dans votre implémentation.

3 - Cherchez à mettre en évidence les problèmes posés par l'affectation d'objets du type vect, tel qu'il est défini dans l'exemple du paragraphe3.2.b.

4 - Ecrivez un programme permettant de mettre en évidence l'ordre d'appel des constructeurs et des destructeurs dans la situation du paragraphe 5.2 (objets membre), ainsi que dans celle de la seconde remarque (objet comportant plusieurs objets membre). Expérimentez également la situation d'objets membre d'objets membre.

5 - (C) Ecrivez une classe nommée pile_entier permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un tableau d'entiers alloués dynamiquement. La classe comportera les fonctions membre suivantes :

pile_entier (int n)

constructeur allouant dynamiquement un emplacement de n entiers,

pile_entier ( )

constructeur sans argument allouant par défaut un emplacement de vingt entiers,

~ pile_entier ( )

destructeur

void empile (int p)

ajoute l'entier p sur la pile,

int depile ( )

fournit la valeur de l'entier situé en haut de la pile, en le supprimant de la pile,

int pleine ( )

fournit 1 si la pile est pleine, 0 sinon,

int vide ( )

fournit 1 si la pile est vide, 0 sinon.

6 - (C) Ecrivez une fonction main, utilisant des objets **automatiques** et **dynamiques** du type pile_entier défini précédemment.

7 - Mettez en évidence les problèmes posés par des déclarations de la forme :

```
pile_entier a(10) ;
pile_entier b = a ;
```

8 - (C) Ajoutez à la classe pile_entier le constructeur de recopie permettant de régler les problèmes précédents.

III. LES FONCTIONS AMIES

La P.O.O. pure impose l'encapsulation des données. Nous avons vu comment la mettre en œuvre en C++ : les membres privés d'une classe ne sont accessibles qu'aux fonctions membre de cette classe (publique ou privée[1], mais seules les fonctions publiques peuvent être appelées "de l'extérieur").

D'autre part, nous avons vu qu'en C++ "l'unité de protection" est la classe, c'est-à-dire qu'une même fonction membre peut accéder à tous les objets de sa classe. Nous en avons rencontré un exemple dans la fonction coincide (examen de la coïncidence de deux objets de type point) du paragraphe 4 du chapitre VI.

En revanche, ce même principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe. Or, il existe des circonstances où une telle contrainte s'avère gênante. Supposez, par exemple, que vous ayez défini une classe vecteur (de taille fixe ou variable, peu importe !) et une classe matrice. Il est probable que vous souhaiterez alors définir une fonction permettant de calculer le produit d'une matrice par un vecteur. Or, avec ce que nous connaissons actuellement de C++, nous ne pourrions définir cette fonction, ni comme fonction membre de la classe vecteur, ni comme fonction membre de la classe matrice, et encore moins comme fonction indépendante (c'est-à-dire membre d'aucune classe).

Bien entendu, vous pourriez toujours "vous en sortir" en rendant publiques les données de vos deux classes, mais vous perdriez alors le bénéfice de leur protection. Vous pourriez également introduire, dans les deux classes, des fonctions publiques permettant d'accéder aux données, mais vous seriez alors pénalisé en temps d'exécution...

En fait, la notion de fonction amie[2] va apporter à ce problème une solution intéressante se présentant sous la forme d'un compromis entre encapsulation formelle des données privées et des données publiques. En effet, il est possible, lors de la définition d'une classe, d'y déclarer qu'une ou plusieurs fonctions (extérieures à la classe) sont des "amies" ; une telle déclaration d'amitié les autorise alors à accéder aux données privées, au même titre que n'importe quelle fonction membre.

L'avantage de cette méthode est de permettre le contrôle des accès au niveau de la classe concernée : on ne peut pas "s'imposer comme fonction amie d'une classe" si cela n'a pas été prévu dans la classe. Nous verrons toutefois qu'en pratique la protection est un peu moins efficace qu'il n'y paraît, dans la mesure où, dans certains cas, une fonction peut se faire passer pour une autre !

Il existe plusieurs situations d'amitiés :

---

1 - Comme nous l'avons déjà dit depuis la version 1.2, il existe également le statut protégé (protected). Nous en parlerons dans le chapitre XI consacré à l'héritage ; pour l'instant, vous pouvez considérer que les membres protégés sont traités comme les membres privés.

2 - "Friend", en anglais.

- fonction indépendante, amie d'une classe,
- fonction membre d'une classe, amie d'une autre classe,
- fonction amie de plusieurs classes,
- toutes les fonctions membre d'une classe, amies d'une autre classe.

La première nous servira à vous présenter les principes généraux de déclaration, définition et utilisation d'une fonction amie. Nous examinerons ensuite en détail chacune de ces situations d'amitié. Enfin, nous verrons l'incidence de l'existence éventuelle de fonctions amies sur l'exploitation d'une classe.

1. EXEMPLE DE FONCTION INDÉPENDANTE AMIE D'UNE CLASSE

Dans le paragraphe 4 du chapitre VI, nous vous avons proposé une fonction coincide examinant la "coïncidence" de deux objets de type point ; pour ce faire, nous en avions fait une fonction membre de la classe point. Nous vous proposons ici de résoudre le même problème, en faisant cette fois de la fonction coincide, une fonction indépendante amie de la classe point.

Tout d'abord, il nous faut introduire dans la classe point, la déclaration d'amitié appropriée, à savoir :

```
friend int coincide (point, point) ;
```

Il s'agit précisément du prototype de la fonction coincide, précédé du mot clé friend. Naturellement, ici, nous avons prévu que coincide recevrait deux arguments de type point (cette fois, il ne s'agit plus d'une fonction membre : elle ne recevra donc pas d'argument implicite - this - correspondant à l'objet l'ayant appelé).

L'écriture de la fonction coincide ne pose aucun problème particulier.

Voici un exemple de programme :

```
#include <iostream.h>
class point
   { int x, y ;
   public :
     point (int abs=0, int ord=0)         // un constructeur ("inline")
             { x=abs ; y=ord ; }
        // déclaration fonction amie (indépendante) nommée coincide
     friend int coincide (point, point) ;
   } ;
int coincide (point p, point q)           // définition de coincide
{   if ((p.x == q.x) && (p.y == q.y)) return 1 ;
                                  else return 0 ;
}
main()                                    // programme d'essai
{  point a(1,0), b(1), c ;
   if (coincide (a,b)) cout << "a coincide avec b \n" ;
                 else cout << "a et b sont différents \n" ;
   if (coincide (a,c)) cout << "a coincide avec c \n" ;
                 else cout << "a et c sont différents \n" ;
}
```

```
a coincide avec b
a et c sont différents
```

Exemple de fonction indépendante (coincide), amie de la classe point

Remarques :

1) L'emplacement de la déclaration d'amitié, au sein de la classe point, est absolument indifférent.

2) Il n'est pas nécessaire de déclarer la fonction amie dans la fonction ou dans le fichier source où on l'utilise, dans la mesure où elle est déjà obligatoirement déclarée dans la classe concernée. Ceci reste valable dans le cas (usuel) où la classe a été compilée séparément puisqu'il faudra alors en introduire la déclaration (généralement par #include). Néanmoins, une déclaration superflue de la fonction amie ne constituerait pas une erreur.

3) Comme nous l'avons déjà fait remarquer, nous n'avons plus ici d'argument implicite (this). Ainsi, contrairement à ce qui se produisait dans notre fonction coincide du paragraphe 4 du chapitre VI, notre fonction coincide est maintenant parfaitement symétrique. Nous retrouverons le même phénomène lorsque, pour surdéfinir un opérateur binaire, nous pourrons choisir entre une fonction membre (dissymétrique) ou une fonction amie (symétrique).

4) Ici, les deux arguments de coincide sont transmis par valeur. Ils pourraient l'être par référence ; notez que, dans le cas d'une fonction membre, l'objet appelant la fonction est d'office transmis par référence (sous la forme de this).

5) Généralement, une fonction amie d'une classe possédera un ou plusieurs arguments ou une valeur de retour du type de cette classe (c'est cela qui justifiera son besoin d'accès aux membres privés des objets correspondants). Ce n'est toutefois pas là une obligation[3] : on pourrait imaginer une fonction ayant besoin d'accéder aux membres privés d'objets locaux à cette fonction...

6) Lorsqu'une fonction amie d'une classe fournit une valeur de retour du type de cette classe, il est fréquent que cette valeur soit celle d'un objet local à la fonction. Dans ce cas, il est alors impératif que sa transmission ait lieu par valeur ; dans le cas d'une transmission par référence (ou par adresse), la fonction appelante recevrait l'adresse d'un emplacement mémoire qui aurait été libéré à la sortie de la fonction. Ce phénomène a déjà été évoqué dans le paragraphe 6 du chapitre VI.

2. LES DIFFÉRENTES SITUATIONS D'AMITIÉ

Nous venons d'examiner le cas d'une fonction indépendante amie d'une classe. Celle-ci peut être résumée par le schéma suivant :

```
class point
{                                          int coincide (point ..., point
...)
   // partie privée                        {  // on a accès ici aux membres
privés
```

3 - Mais cela en sera une dans le cas des opérateurs surdéfinis.

```
      .....                                        // de tout objet de type point
    // partie publique                          }
       friend int coincide (point, point) ;
       .....
} ;
```

Fonction indépendante (coincide) amie d'une classe (point)

Rappelons que, bien qu'ici nous l'ayons placée dans la partie publique de point, la déclaration d'amitié peut figurer n'importe où dans la classe.

D'autres situations d'amitié sont possibles ; basées sur le même principe, elles peuvent conduire à des déclarations d'amitié très légèrement différentes. Nous allons maintenant les passer en revue.

2.1 Fonction membre d'une classe, amie d'une autre classe

Il s'agit un peu d'un cas particulier de la situation précédente. En fait, il suffit simplement, dans la déclaration d'amitié, de préciser la classe à laquelle appartient la fonction concernée, à l'aide de l'opérateur de résolution de portée (::).

Par exemple, supposons que nous ayons à définir deux classes nommées A et B et que, dans B, nous ayons besoin d'une fonction membre f, de prototype :

```
int f(char, A) ;
```

Si, comme il est probable, f doit pouvoir accéder aux membres privés de A, elle sera déclarée amie, au sein de la classe par :

```
friend int B::f(char, A) ;
```

Voici un schéma récapitulatif de la situation :

```
class A                                    class B
{                                          {
     // partie privée                            .....
        .....                                   int f (char, A) ;
     // partie publique                         .....
        friend int B::f (char, A) ;        }
        .....                              int B::f (char ..., A ...)
} ;                                        {  // on a accès ici aux membres
privés
                                              // de tout objet de type A
                                           }
```

Fonction (f) d'une classe (B), amie d'une autre classe (A)

Remarque importante :

Pour compiler convenablement les déclarations d'une classe A contenant une déclaration d'amitié telle que :

```
friend int B::f(char, A) ;
```

le compilateur a besoin de connaître les caractéristiques de B ; cela signifie que la déclaration de B (mais pas nécessairement la définition de ses fonctions membre) devra avoir été compilée avant celle de A.

En revanche, pour compiler convenablement la déclaration :

```
int f(char, A)
```

figurant au sein de la classe B, le compilateur n'a pas besoin de connaître précisément les caractéristiques de A. Il lui suffit de savoir qu'il s'agit d'une classe. Comme d'après ce qui vient d'être dit, la déclaration de B n'a pu apparaître avant, on fournira l'information voulue au compilateur en faisant précéder la déclaration de A de :

```
class A ;
```

Bien entendu, la compilation de la définition de la fonction f nécessite (en général[4]) la connaissance des caractéristiques des classes A et B ; leurs déclarations devront donc apparaître avant.

A titre indicatif, voici une façon de compiler nos deux classes A et B et la fonction f :

```
class A ;
class B
{     .....
      int f(char, A) ;
      .....
} ;
class A
{     .....
      friend int B::f(char, A) ;
      .....
} ;
int B::f(char..., A...)
{     .....
}
```

Remarque :

Si l'on a besoin de "déclarations d'amitiés croisées" entre fonctions de deux classes différentes, la seule façon d'y parvenir consiste à déclarer au moins une des classes amie de l'autre (comme nous apprendrons à le faire dans le paragraphe 2.3).

2.2 Fonction amie de plusieurs classes

Rien n'empêche qu'une même fonction (qu'elle soit indépendante ou fonction membre) fasse l'objet de déclarations d'amitié dans différentes classes. Voici un exemple d'une fonction amie de deux classes A et B.

```
class A
{
      // partie privée
         .....
      // partie publique
         friend void f(A, B) ;
;
         .....
} ;
```

```
class B
{
      // partie privée
         .....
            // partie publique
               friend void f(A, B)

         .....
} ;
```

4 - Une exception aurait lieu pour B si f n'accédait à aucun de ses membres (ce qui serait surprenant). Il en irait de même pour A si aucun argument de ce type n'apparaissait dans f et si cette dernière n'accédait à aucun membre de A (ce qui serait tout aussi surprenant).

```
void f(A..., B...)
{  // on a accès ici aux membres privés
   // de n'importe quel objet de type A ou B
}
```

Fonction indépendante (f) amie de deux classes (A et B)

Remarque

Ici, la déclaration de A peut être compilée sans celle de B, en la faisant précéder de la déclaration :

```
class B ;
```

De même, la déclaration de B peut être compilée sans celle de A, en la faisant précéder de la déclaration :

```
class A ;
```

Si l'on compile en même temps les deux déclarations de A et B, il faudra utiliser l'une des deux déclarations citées (class A si B figure avant A, class B sinon).

Bien entendu, la compilation de la définition de f nécessitera généralement les déclarations de A et de B.

2.3 Toutes les fonctions d'une classe sont amies d'une autre classe

C'est une généralisation du cas évoqué dans le paragraphe 2.1. On pourrait d'ailleurs effectuer autant de déclarations d'amitié qu'il y a de fonctions concernées. Il est plus simple, dans ce cas, d'effectuer une déclaration globale. Ainsi, pour dire que toutes les fonctions membre de la classe B sont amies de la classe A, on placera, dans la classe A, la déclaration :

```
friend class B ;
```

Remarques :

1) Cette fois, pour compiler la déclaration de la classe A, il suffira de la faire précéder de :

```
class B ;
```

2) Ce type de déclaration d'amitié évite d'avoir à fournir les en-têtes des fonctions concernées.

3. EXEMPLE

Nous vous proposons ici de résoudre le problème évoqué en introduction, à savoir réaliser une fonction permettant de déterminer le produit d'un vecteur (objet de classe vect) par une matrice (objet de classe matrice). Par souci de simplicité, nous avons limité les fonctions membre à :

- un constructeur pour vect et pour matrice,
- une fonction d'affichage (affiche) pour matrice.

Nous vous fournissons deux solutions basées sur l'emploi d'une fonction amie nommée prod :

- prod est indépendante et amie des deux classes vect et matrice,
- prod est membre de matrice et amie de la classe vect.

3.1 Fonction amie indépendante

```
#include <iostream.h>
class matrice ;         // pour pouvoir compiler la déclaration de vect

       // *********** La classe vect ******************
class vect
{
    double v[3] ;       // vecteur à 3 composantes
  public :
    vect (double v1=0, double v2=0, double v3=0)   // constructeur
      { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ;
      }
    friend vect prod (matrice, vect) ;   // prod = fonction amie indépendante
    affiche ()
      { int i ;
        for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
        cout << "\n" ;
      }
} ;

       // *********** La classe matrice *****************
class matrice
{   double mat[3] [3] ;        // matrice 3 X 3
  public :
    matrice (double t[3][3])    // constructeur, à partir d'un tableau 3 x 3
      { int i ; int j ;
        for (i=0 ; i<3 ; i++)
          for (j=0 ; j<3 ; j++)
            mat[i] [j] = t[i] [j] ;
      }
    friend vect prod (matrice, vect) ;   // prod = fonction amie indépendante
} ;
       // ********** La fonction prod *****************
vect prod (matrice m, vect x)
{  int i, j ;
   double som ;
   vect res ;     // pour le résultat du produit
   for (i=0 ; i<3 ; i++)
     { for (j=0, som=0 ; j<3 ; j++)
         som += m.mat[i] [j] * x.v[j] ;
       res.v[i] = som ;
     }
   return res ;
}
       // ********** Un petit programme de test *********
main()
```

```
{
   vect w (1,2,3) ;
   vect res  ;
   double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
   matrice a =  tb  ;
   res = prod(a, w) ;
   res.affiche () ;
}
```

```
14 32 50
```

Produit d'une matrice par un vecteur à l'aide d'une fonction indépendante amie des deux classes

3.2 Fonction amie, membre d'une classe

```
#include <iostream.h>
       // ********* Déclaration de la classe matrice ************
class vect ;                    // pour pouvoir compiler correctement
class matrice
{   double mat[3] [3] ;         // matrice 3 X 3
  public :
    matrice (double t[3][3])    // constructeur, à partir d'un tableau 3 x 3
      { int i ; int j ;
        for (i=0 ; i<3 ; i++)
          for (j=0 ; j<3 ; j++)
            mat[i] [j] = t[i] [j] ;
      }
    vect prod (vect) ;          // prod = fonction membre (cette fois)
} ;
       // ********* Déclaration de la classe vect **************
class vect
{   double v[3] ;      // vecteur à 3 composantes
  public :
    vect (double v1=0, double v2=0, double v3=0)   // constructeur
      { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }
    friend vect matrice::prod (vect) ;             // prod = fonction amie
    affiche ()
      { int i ;
        for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
        cout << "\n" ;
      }
} ;
        // ********* Définition de la fonction prod ************
vect matrice::prod (vect x)
{  int i, j ;
   double som ;
   vect res ;     // pour le résultat du produit
   for (i=0 ; i<3 ; i++)
     { for (j=0, som=0 ; j<3 ; j++)
         som += mat[i] [j] * x.v[j] ;
```

```
        res.v[i] = som ;
      }
   return res ;
}
        // ********** Un petit programme de test *********
main()
{
   vect w (1,2,3) ;
   vect res  ;
   double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
   matrice a =  tb  ;
   res = a.prod (w) ;
   res.affiche () ;
}
14 32 50
```

Produit d'une matrice par un vecteur à l'aide d'une fonction membre amie d'une autre classe

4. EXPLOITATION DE CLASSES DISPOSANT DE FONCTIONS AMIES

Comme nous l'avons déjà mentionné dans le chapitre V, les classes seront généralement compilées séparément. Leur utilisation se fera à partir d'un module objet contenant les fonctions membre de la classe et d'un fichier en-tête contenant la déclaration de la classe. Bien entendu, il est toujours possible de regrouper plusieurs classes dans un même module objet et éventuellement dans un même fichier en-tête.

Dans tous les cas, cette compilation séparée des classes permet d'en assurer la réutilisabilité : le "client" (qui peut éventuellement être le concepteur de la classe) ne peut pas intervenir sur le contenu des objets de cette classe.

Que deviennent ces possibilités lorsque l'on utilise des fonctions amies ? En fait, s'il s'agit de fonctions amies, membres d'une classe, rien n'est changé (en dehors des éventuelles déclarations de classes nécessaires à son emploi). En revanche, s'il s'agit d'une fonction indépendante, il faudra bien voir que si l'on souhaite en faire un module objet séparé, on court le risque de voir l'utilisateur de la classe violer le principe d'encapsulation.

En effet, dans ce cas, l'utilisateur d'une classe disposant d'une fonction amie peut toujours ne pas incorporer la fonction amie à l'édition de liens et fournir lui-même une autre fonction de même en-tête et, par suite, accéder comme il l'entend aux données privées...

Ce risque d'"effet caméléon" doit être nuancé par le fait qu'il s'agit d'une action délibérée (demandant un certain travail), et non pas d'une simple étourderie...

IX. LA SURDÉFINITION D'OPÉRATEURS

Nous avons vu, dans le chapitre IV, comment C++ autorise la "surdéfinition" de fonctions, qu'il s'agisse de fonctions membre ou de fonctions indépendantes. Rappelons que cette technique consiste à attribuer le même nom à des fonctions différentes ; lors d'un appel, le choix de la "bonne fonction" est effectué par le compilateur, suivant le nombre et le type des arguments.

Mais C++ permet également, dans certaines conditions, de surdéfinir des opérateurs. En fait, le langage C, comme beaucoup d'autres, réalise déjà la surdéfinition de certains opérateurs. Par exemple, dans une expression telle que :

```
a + b
```

le symbole + peut désigner, suivant le type de a et b :

- l'addition de deux entiers,
- l'addition de deux réels (float),
- l'addition de deux réels double précision (double),
- etc.

De la même manière, le symbole * peut, suivant le contexte, représenter la multiplication d'entiers ou de réels ou une "indirection" (comme dans a = * adr).

En C++, vous pourrez surdéfinir n'importe quel opérateur existant (unaire ou binaire), dans la mesure où il porte sur au moins un objet[1]. Il s'agit là d'une technique fort puissante puisqu'elle va vous permettre de créer, par le biais des classes, des types à part entière, c'est-à-dire munis, comme les types de base, d'opérateurs parfaitement intégrés ; la notation opératoire qui en découlera aura l'avantage sur une notation fonctionnelle (par appel de fonction) d'être beaucoup plus concise et (du moins si l'on s'y prend "intelligemment" !) plus lisible.

Par exemple, si vous définissez une classe complexe destinée à représenter des nombres complexes, il vous sera possible de donner une signification à des expressions telles que :

```
a + b     a - b     a * b     a/b
```

1 - Cette restriction signifie simplement qu'il ne sera pas possible de surdéfinir les opérateurs portant sur les différents types de base.

a et b étant des objets de type complexe[2]. Pour cela, vous "surdéfinirez" les opérateurs +, -, * et / en spécifiant le rôle exact que vous souhaitez leur attribuer. Cette définition se déroulera comme celle d'une fonction à laquelle il suffira simplement d'attribuer un nom spécial permettant de spécifier qu'il s'agit en fait d'un opérateur. Autrement dit, la surdéfinition d'opérateurs en C++ consistera simplement en l'écriture de nouvelles fonctions surdéfinies.

Nous commencerons par vous présenter cette nouvelle technique qu'est la surdéfinition d'opérateurs. Puis nous verrons quelles en sont les possibilités et les limites.

Nous étudierons ensuite deux exemples importants de surdéfinition d'opérateurs : = et []. Certes, il ne s'agira que d'applications, mais elles montreront qu'à partir du moment où l'on souhaite donner à de tels opérateurs une signification naturelle et acceptable dans un contexte de classe, un certain nombre de précautions doivent être prises. En particulier, nous verrons comment la surdéfinition de l'affectation permet de régler le problème déjà rencontré, à savoir celui des objets comportant des pointeurs sur des emplacements dynamiques.

Enfin, nous examinerons comment prendre en charge la gestion de la mémoire en surdéfinissant les opérateurs new et delete.

1. LE MÉCANISME DE LA SURDÉFINITION D'OPÉRATEUR

Considérons une classe point :

```
class point
{     int x, y ;
      .....
```

et supposons que nous souhaitons définir l'opérateur + afin de donner une signification à une expression telle que a + b, lorsque a et b sont de type point. Ici, nous conviendrons que la "somme" de deux points est un point dont les coordonnées sont la somme de leurs coordonnées[3].

La convention adoptée par C++ pour surdéfinir cet opérateur + consiste à définir une fonction de nom :

```
operator +
```

On y trouve le mot clé operator suivi de l'opérateur concerné (dans le cas présent, il ne serait pas obligatoire de prévoir un espace car, en C, + sert de "séparateur").

Ici, notre fonction operator + doit disposer de deux arguments de type point et fournir une valeur de retour du même type. En ce qui concerne sa nature, cette fonction peut, à notre gré, être une fonction membre de la classe concernée ou une fonction indépendante ; dans ce dernier cas, il s'agira généralement d'une fonction amie, dans la mesure où il lui faudra pouvoir accéder aux membres privés de la classe.

Examinons ici les deux solutions, en commençant par celle qui est la plus "naturelle", à savoir la fonction amie.

---

2 - Une notation fonctionnelle conduirait à des choses telles que : somme (a,b) ou a.somme(b) suivant que l'on utilise une fonction amie ou une fonction membre

3 - Nous aurions pu tout aussi bien prendre ici l'exemple de la classe complexe évoquée en introduction. Nous préférons cependant choisir un exemple dans lequel la signification de l'opérateur n'a pas un caractère aussi évident. En effet, n'oubliez pas que n'importe quel symbole opérateur peut, au bout du compte, se voir attribuer n'importe quelle signification !

1.1 Surdéfinition d'opérateur avec une fonction amie

Le prototype de notre fonction operator + sera :

```
point operator + (point, point) ;
```

Ses deux arguments correspondront aux opérandes de l'opérateur + lorsqu'il sera appliqué à des valeurs de type point.

Le reste du travail est classique :

- déclaration d'amitié au sein de la classe point,
- définition de la fonction.

Voici un exemple de programme montrant la définition et l'utilisation de notre "opérateur d'addition de points".

```
#include <iostream.h>
class point
   {  int x, y ;
   public :
      point (int abs=0, int ord=0) { x=abs ; y=ord ;}  // constructeur
      friend point operator+ (point, point) ;
      void affiche () { cout << "coordonnées : " << x << " " << y << "\n" ; }
    } ;
point operator + (point a, point b)
{  point p ;
   p.x = a.x + b.x ; p.y = a.y + b.y ;
   return p ;
}
main()
{  point a(1,2) ; a.affiche() ;
   point b(2,5) ; b.affiche() ;
   point c ;
   c = a+b ;      c.affiche() ;
   c = a+b+c ;    c.affiche() ;
}
```

```
coordonnées : 1 2
coordonnées : 2 5
coordonnées : 3 7
coordonnées : 6 14
```

Surdéfinition de l'opérateur + pour des objets de type point,
en employant une fonction amie

Remarques :

1) Une expression telle que a + b est en fait interprétée par le compilateur comme l'appel :

```
operator + (a, b)
```

Bien que cela ne présente guère d'intérêt, nous pourrions écrire :

```
c = operator + (a, b)
```

au lieu de c = a + b.

2) Une expression telle que a + b + c est évaluée en tenant compte des règles de priorité et d'associativité "habituelles" de l'opérateur +. Nous reviendrons plus loin sur ce point d'une manière générale. Pour l'instant, notez simplement que cette expression est évaluée comme :

```
(a + b) + c
```

c'est-à-dire en utilisant la notation fonctionnelle :

```
operator + (operator + (a, b), c)
```

1.2 Surdéfinition d'opérateur avec une fonction membre

Cette fois, le premier opérande de notre opérateur, correspondant au premier argument de la fonction operator + précédente, va se trouver transmis implicitement : ce sera l'objet ayant appelé la fonction membre. Par exemple, une expression telle que a + b sera alors interprétée par le compilateur comme :

```
a.operator + (b)
```

Le prototype de notre fonction membre operator + sera donc :

```
point operator + (point)
```

Voici comment pourrait être adapté dans ce sens l'exemple précédent :

```
#include <iostream.h>
class point
   {  int x, y ;
   public :
      point (int abs=0, int ord=0) { x=abs ; y=ord ;}  // constructeur
      point operator + (point) ;
      void affiche () { cout << "coordonnées : " << x << " " << y << "\n" ; }
    } ;
point point::operator + (point a)
{  point p ;
   p.x = x + a.x ; p.y = y + a.y ;
   return p ;
}
main()
{  point a(1,2) ; a.affiche() ;
   point b(2,5) ; b.affiche() ;
```

```
    point c ;
    c = a+b ;       c.affiche() ;
    c = a+b+c ;     c.affiche() ;
}
```

```
coordonnées : 1 2
coordonnées : 2 5
coordonnées : 3 7
coordonnées : 6 14
```

Surdéfinition de l'opérateur + pour des objets de type point,
en employant une fonction membre

Remarques :

1) Cette fois, la définition de la fonction operator + fait apparaître une dissymétrie entre les deux opérandes. Par exemple, le membre x est noté x pour le premier opérande (argument implicite) et a.x pour le second. Cette dissymétrie peut parfois inciter l'utilisateur à choisir une fonction amie plutôt qu'une fonction membre. Il faut toutefois se garder de décider trop vite dans ce domaine. Nous y reviendrons un peu plus loin.

2) Ici, l'affectation :

```
c = a + b ;
```

est interprétée comme :

```
c = a.operator + (b) ;
```

Quant à l'affectation :

```
c = a + b + c ;
```

le langage C++ ne précise pas exactement son interprétation. Certains compilateurs créeront un objet temporaire t :

```
t = a.operator + (b) ;
c = t.operator + (c) ;
```

D'autres procéderont ainsi, en transmettant comme adresse de l'objet appelant operator +, l'adresse de l'objet renvoyé par l'appel précédent :

```
c = (a.operator + (b)).operator + (c) ;
```

On peut détecter le choix fait par un compilateur en faisant afficher toutes les créations d'objets (en n'oubliant pas d'introduire un constructeur de recopie prenant la place du constructeur par défaut).

1.3 Opérateurs et transmission par référence

Dans nos deux précédents exemples, la transmission des arguments (deux pour une fonction amie, un pour une fonction membre) et de la valeur de retour de operator + se faisait par valeur[4].

Bien entendu, en particulier dans le cas d'objets de grande taille, on peut envisager de faire appel au transfert par référence. Par exemple, le prototype de la fonction amie operator + pourrait être :

```
point operator + (point & a, point & b) ;
```

En revanche, la transmission par référence poserait un problème si on cherchait à l'appliquer à la valeur de retour. En effet, le point p est créé localement dans la fonction ; il sera donc détruit dès la fin de son exécution. Dans ces conditions, employer la transmission par référence reviendrait à transmettre l'adresse d'un emplacement de mémoire libéré.

Certes, ici, nous utilisons immédiatement la valeur de p, dès le retour dans la fonction main (ce qui est généralement le cas avec un opérateur) ; néanmoins, nous ne pouvons faire aucune hypothèse sur la manière dont une implémentation donnée libère un emplacement mémoire : elle peut simplement se contenter de "noter" qu'il est disponible, auquel cas son contenu reste "valable" pendant un... certain temps ; elle peut, au contraire le "mettre à zéro"... La première situation est certainement la pire puisqu'elle peut donner l'illusion que "cela marche" !

Pour éviter la recopie de cette valeur de retour, on pourrait songer à allouer dynamiquement l'emplacement de p. Généralement, cela prendra plus de temps que sa recopie ultérieure et, de plus, compliquera quelque peu le programme (il faudra libérer convenablement l'emplacement en question et on ne pourra le faire qu'en dehors de la fonction !).

Si l'on cherche à protéger contre d'éventuelles modifications un argument transmis par référence, on pourra toujours faire appel au mot clé const ; par exemple, l'en-tête de operator + pourrait être :

```
point operator + (const point& a, const point& b) ;
```

Naturellement, si l'on utilise const dans le cas d'objets comportant des pointeurs sur des parties dynamiques, seuls ces pointeurs seront ainsi "protégés" ; les parties dynamiques resteront modifiables.

2. LES POSSIBILITÉS ET LES LIMITES DE LA SURDÉFINITION D'OPÉRATEUR

Nous venons de voir un exemple de surdéfinition de l'opérateur binaire +, lorsqu'il reçoit deux opérandes de type point, et ceci de deux façons : comme fonction amie, comme fonction membre.

Nous vous proposons de voir maintenant ce qu'il est possible de faire d'une manière générale.

2.1 Il faut se limiter aux opérateurs existants...

4 - Rappelons que la transmission de l'objet appelant une fonction membre se fait par référence.

Le symbole suivant le mot clé operator doit obligatoirement être un opérateur déjà défini pour les types de base. Il n'est donc pas possible de créer de nouveaux symboles. Nous verrons d'ailleurs que certains opérateurs ne peuvent pas être redéfinis du tout (c'est le cas de .) et que d'autres imposent quelques contraintes supplémentaires.

Il faut conserver la pluralité (unaire, binaire) de l'opérateur initial. Ainsi, vous pourrez surdéfinir un opérateur + unaire ou un opérateur + binaire, mais vous ne pourrez pas définir de = unaire ou de ++ binaire.

Lorsque plusieurs opérateurs sont combinés au sein d'une même expression (qu'ils soient surdéfinis ou non), ils conservent leur priorité relative et leur associativité. Par exemple, si vous surdéfinissez les opérateurs binaires + et * pour le type complexe, l'expression suivante (a, b et c étant supposés du type complexe) :

```
a * b + c
```

sera interprétée comme :

```
(a * b) + c
```

De telles règles peuvent, a priori, vous paraître restrictives. En fait, vous verrez à l'usage qu'elles sont encore très larges et qu'il est facile de rendre un programme incompréhensible en abusant de la surdéfinition d'opérateurs.

Le tableau de la page suivante précise les opérateurs surdéfinissables (en fait, tous sauf .) et il rappelle leur priorité relative et leur associativité. Notez la présence :

- de l'opérateur de "cast" ; nous en parlerons plus en détail dans le chapitre X. Nous y verrons qu'il peut s'appliquer à la conversion d'une classe dans un type de base ou à la conversion d'une classe dans une autre classe.
- des opérateurs new et delete : avant la version 2.0, ces opérateurs étaient traités comme des exceptions puisqu'on ne pouvait pas les surdéfinir pour une classe en particulier ; on ne pouvait en modifier la signification que d'une façon globale. Depuis la version 2.0, ils sont surdéfinissables au même titre que les autres. Nous en parlerons dans le paragraphe 6.

PLURALITÉ	OPÉRATEURS	ASSOCIATIVITÉ
Binaire	() [] ->	->
Unaire	+ - ++ -- ! ~ * & <- new new [] delete (cast)	
Binaire	* / %	->
Binaire	+ -	->
Binaire	<< >>	->
Binaire	< <= > >=	->
Binaire	== !=	->
Binaire	&	->
Binaire	^	->
Binaire	||	->
Binaire	&&	->
Binaire	|	->
Binaire	= += -= *= /= %= &= ^= |= <<= >>=	<-
Binaire	,	->

Les opérateurs surdéfinissables en C++ (classés par priorité décroissante)

(1) S'il n'est pas surdéfini, il possède une signification par défaut.
(2) Depuis la version 2.0 seulement.
(3) Doit être défini comme fonction membre.
(4) A un "niveau global" avant la version 2.0. Depuis la version 2.0, il peut, en outre, être surdéfini pour une classe ; dans ce cas, il doit l'être comme fonction membre.
(5) Jusqu'à la version 3, on ne peut pas distinguer entre les notations "pré" et "post". Depuis la version 3, ces opérateurs (lorsqu'ils sont définis de façon unaire) correspondent à la notation "pré" ; mais il en existe une définition binaire (avec deuxième opérande fictif de type int) qui correspond à la notation "post".
(6) On distingue bien new de new[].

2.2 ... au contexte de classe

On ne peut surdéfinir un opérateur que s'il comporte au moins un argument (implicite ou non) de type classe. Autrement dit, il doit s'agir :

- Soit d'une fonction membre : dans ce cas, elle comporte à coup sûr un argument (implicite) de type classe, à savoir l'objet l'ayant appelé. S'il s'agit d'un opérateur unaire, elle ne comportera aucun argument explicite. S'il s'agit d'un opérateur binaire, elle comportera un argument explicite auquel

aucune contrainte de type n'est imposée (dans nos précédents exemples, il s'agissait du même type que la classe elle-même, mais il pourrait s'agir d'un autre type classe ou même d'un type de base).

- Soit d'une fonction indépendante ayant au moins un argument de type classe. En général, il s'agira d'une fonction amie.

Cette règle garantit à coup sûr l'impossibilité de surdéfinir un opérateur portant sur des types de base (imaginez ce que serait un programme dans lequel on pourrait changer la signification de 3 + 5 ou de * adr !). Une exception a lieu, cependant, pour les seuls opérateurs new et delete dont la signification peut être modifiée de manière globale (pour **tous** les objets et les types de base) ; nous en reparlerons dans le paragraphe 6.

De plus, certains opérateurs doivent obligatoirement être définis comme membres d'une classe. Il s'agit de [], ( ), -> [5], new[6] et delete[6].

2.3 ... et ne pas faire d'hypothèse sur la "signification" d'un opérateur

Comme nous avons déjà eu l'occasion de l'indiquer, vous êtes totalement libre d'attribuer à un opérateur surdéfini la signification que vous désirez. Cette liberté n'est limitée que par le bon sens qui doit vous inciter à donner à un symbole une signification relativement naturelle : par exemple + pour la somme de deux complexes, plutôt que -, * ou [].

Mais, qui plus est, vous ne retrouverez pas, pour les opérateurs surdéfinis, les liens qui existent entre certains opérateurs de base. Par exemple, si a et b sont de type int :

```
a += b
```

est équivalent à :

```
a = a + b
```

Autrement dit, le rôle de l'opérateur de base **+ =** se déduit du rôle de l'opérateur + et de celui de l'opérateur **=** . En revanche, si vous surdéfinissez, par exemple, l'opérateur + et l'opérateur = lorsque leurs deux opérandes sont de type complexe, vous n'aurez pas, pour autant, défini la signification de += lorsqu'il aura deux opérandes de type complexe. De plus, dans ce cas, vous pourrez très bien surdéfinir += pour qu'il ait une signification différente de celle attendue ; naturellement, cela n'est pas conseillé...

De même, et de façon peut-être plus surprenante, C++ ne fait aucune hypothèse sur la commutativité éventuelle d'un opérateur surdéfini (alors que, rappelons-le, il en fait en ce qui concerne sa priorité relative et son associativité). Cette remarque est lourde de conséquences. Supposez, par exemple, que vous ayez surdéfini l'opérateur + lorsqu'il a comme opérandes un complexe et un double (dans cet ordre) ; son prototype pourrait être :

```
complexe operator + (complexe, double) ;
```

Eh bien, si ceci vous permet de donner un sens à une expression telle que (a étant complexe) :

```
a + 3.5
```

cela ne permet pas pour autant d'interpréter :

```
3.5 + a
```

---

5 - Il n'est surdéfinissable que depuis la version 2.0.

6 - Dans le cas où ils ne sont pas surdéfinis de manière globale (ce qui n'est possible que depuis la version 2.0).

Pour ce faire, il aurait fallu, en effet, surdéfinir l'opérateur + lorsqu'il a comme opérandes un double et un complexe avec, par exemple[7], comme prototype :

```
complexe operator + (double, complexe) ;
```

Nous verrons cependant (chapitre X) que les possibilités de conversions définies par l'utilisateur permettront (dans le cas où cela présente de l'intérêt) de simplifier quelque peu les choses. Par exemple, nous verrons que, dans ce cas précis, il suffira de définir l'opérateur + lorsqu'il porte sur deux complexes ainsi que la conversion de double en complexe pour que les expressions de l'une de ces formes aient un sens.

```
double + complexe
complexe + double
float + complexe
complexe + float
```

2.4 Cas des opérateurs + + et --

Jusqu'à la version 2.0 de C++, on ne pouvait pas distinguer l'opérateur ++ en notation préfixée (comme dans ++a) de ce même opérateur en notation postfixée (comme dans a++). Autrement dit, pour un type classe donné, on ne pouvait définir qu'un seul opérateur ++ (operator ++) qui était utilisé dans les deux cas.

Depuis la version 3, on peut définir à la fois un opérateur ++ utilisable en notation préfixée et un autre utilisable en notation postfixée. Plus précisément, si T désigne un type classe quelconque :

- l'opérateur (usuel) d'en-tête T operator ++ () est utilisé en cas de notation préfixée,
- l'opérateur d'en-tête T operator ++ (int) est utilisé en cas de notation postfixée. Notez bien la présence d'un second opérande de type int. Celui-ci est totalement fictif, en ce sens qu'il permet au compilateur de choisir l'opérateur à utiliser mais qu'aucune valeur ne sera réellement transmise lors de l'appel.

Les mêmes remarques s'appliquent à l'opérateur --.

Voici un exemple dans lequel nous avons défini ++ pour qu'il incrémente d'une unité les deux coordonnées d'un point et pour qu'il fournisse comme valeur soit celle du point avant incrémentation dans le cas de la notation postfixée, soit celle du point après incrémentation dans le cas de la notation préfixée.

```
#include <iostream.h>
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  point  operator ++ ()          // notation préfixée
    { x++ ; y++ ; return *this ;
    }
  point  operator ++ (int n)     // notation postfixée
    { point p = *this ;
      x++ ; y++ ;
```

7 - Nous verrons d'ailleurs un peu plus loin que, dans ce cas, on ne pourra pas surdéfinir cet opérateur comme une fonction membre (puisque son premier opérande n'est plus de type classe).

```
        return p ;
      }
  void affiche () { cout << x << " " << y << "\n" ; }
} ;


main()
{ point a1 (2, 5), a2(2, 5), b ;
  b = ++a1 ; cout << "a1 : " ; a1.affiche () ;   // affiche    a1 : 3 6
             cout << "b  : " ; b.affiche () ;    // affiche    b  : 2 5
  b = a2++ ; cout << "a2 : " ; a2.affiche () ;   // affiche    a2 : 3 6
             cout << "b  : " ; b.affiche () ;    // affiche    b  : 3 6
}
```

Exemple de surdéfinition de ++ à la fois en notation préfixée et en notation postfixée

Remarque :

En théorie, il n'est plus possible, depuis la version 3 et donc, en particulier, depuis la norme ANSI, de ne définir qu'un seul opérateur ++ qu'on utiliserait à la fois en notation préfixée et en notation postfixée. En fait, la plupart des compilateurs fournissent dans ce cas un message d'avertissement en utilisant le même opérateur avec les deux notations.

2.5 Les opérateurs = et & ont une signification prédéfinie

Nous avons déjà eu l'occasion d'employer l'opérateur = avec deux opérandes du même type classe. Nous n'avions pas eu, pour cela, à le surdéfinir. Effectivement, en l'absence de surdéfinition explicite, cet opérateur correspond à la recopie des valeurs de son second opérande dans le premier. Nous avons d'ailleurs déjà eu l'occasion de constater que cette simple recopie pouvait s'avérer insatisfaisante dès lors que les objets concernés comportaient des pointeurs sur des emplacements dynamiques. Il s'agit là typiquement d'une situation qui nécessite la surdéfinition de l'opérateur =, dont nous donnerons un exemple dans le paragraphe suivant.

On notera la grande analogie existant entre :

- le constructeur de recopie : s'il n'en existe pas d'explicite, il y a appel d'un constructeur de recopie par défaut,
- l'opérateur d'affectation : s'il n'en existe pas d'explicite, il y a emploi d'un opérateur d'affectation par défaut.

Constructeur de recopie par défaut et opérateur d'affectation par défaut effectuent le même travail : recopie des valeurs de l'objet. Dans le chapitre VII, nous avons signalé que, dans le cas d'objets dont certains membres sont eux-mêmes des objets, le constructeur de recopie par défaut travaillait membre par membre. La même remarque s'applique à l'opérateur d'affectation par défaut : il opère membre par membre[8], ce qui laisse la possibilité d'appeler un opérateur d'affectation explicite, dans le cas où l'un des membres en posséderait un. Ceci peut éviter d'avoir à écrire explicitement un opérateur d'affectation pour des objets sans pointeurs (apparents), mais dont un ou plusieurs membres possèdent, quant à eux, des parties dynamiques.

8 - Là encore, depuis la version 2.0 de C++. Auparavant, il opérait de façon globale ("member-wise copy").

2.6 Les conversions

C et C++ autorisent fréquemment les conversions entre types de base, et ceci de façon explicite ou implicite. Ces possibilités s'étendent aux objets. Par exemple, comme nous l'avons déjà évoqué, si a est de type complexe et si l'opérateur + a été surdéfini pour deux complexes, une expression telle que a + 3.5 pourra prendre un sens :

- soit si l'on a surdéfini l'opérateur + lorsqu'il a un opérande de type complexe et un opérande de type double,
- soit si l'on a défini une conversion de type double en complexe.

Nous avons toutefois préféré regrouper dans un chapitre à part (le suivant) tout ce qui concerne les problèmes de conversion ; c'est là que nous parlerons de la surdéfinition d'un opérateur de "cast".

2.7 Choix entre fonction membre et fonction amie

Comme nous l'avons dit, C++ vous laisse libre de surdéfinir un opérateur à l'aide d'une fonction membre ou d'une fonction indépendante (en général amie). Cela signifie donc que, dans certains cas, vous pourrez vous demander sur quels critères opérer le choix. En toute rigueur, vous ne possédez pas, pour l'instant, toutes les informations nécessaires, compte tenu des possibilités de conversion que nous n'étudierons que dans le prochain chapitre.

Pour l'instant, notez simplement que **si un opérateur doit absolument recevoir un type de base en premier argument, il ne peut pas être défini comme fonction membre** (laquelle reçoit implicitement un premier argument du type de sa classe).

Le cas, déjà évoqué, de l'addition d'un double et d'un complexe (dans cet ordre[9]) peut sembler correspondre à une telle situation. En fait, comme nous l'avons déjà dit dans le paragraphe 2.5, nous verrons, dans le prochain chapitre que celui-ci peut également se résoudre par surdéfinition de l'addition de deux complexes et par la définition de la conversion double -> complexe.

3. EXEMPLE DE SURDÉFINITION DE L'OPÉRATEUR =

Nous avons déjà eu l'occasion d'utiliser une classe vect, correspondant à des "vecteurs dynamiques" :

```
class vect
{     int nelem ;       // nombre d'éléments
      int * adr ;       // adresse
   public :
      vect (int n)      // constructeur
      .....
} ;
```

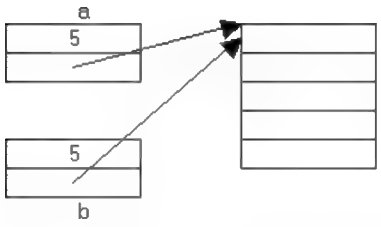
Nous avons vu (paragraphe 3.2 du chapitre VII) qu'alors, si fct était une fonction à un argument de type vect, les instructions suivantes :

```
vect a(5) ;
```

9 - Il ne suffira pas d'avoir surdéfini l'addition d'un complexe et d'un double (laquelle peut se faire par une fonction membre). En effet, comme nous l'avons dit, aucune hypothèse n'est faite par C++ sur l'opérateur surdéfini, en particulier sur sa commutativité !

```
   ...
fct (a) ;
```

posaient problème : l'appel de fct conduisait à la création, par recopie de a, d'un nouvel objet (que nous nommions b) ; nous étions alors en présence de deux objets a et b comportant un pointeur (adr) vers le même emplacement :



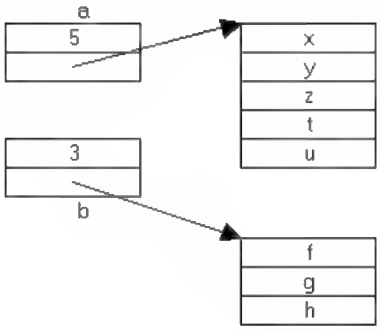
En particulier, si la classe vect possédait (comme c'est souhaitbale !) un destructeur chargé de libérer l'emplacement mémoire en question, on risquait d'aboutir à deux demandes de libération du même emplacement mémoire.

Nous avons vu alors qu'une solution (parmi d'autres) consistait à définir un constructeur de recopie chargé d'effectuer, non seulement la recopie de l'objet lui-même, mais également celle de sa partie dynamique dans un nouvel emplacement (une autre solution consiste à employer un "compteur de références" : nous vous la présenterons dans le paragraphe suivant).

Or, l'affectation d'objets de type vect pose les mêmes problèmes. Ainsi, avec cette déclaration :

```
vect a(5), b(3) ;
```

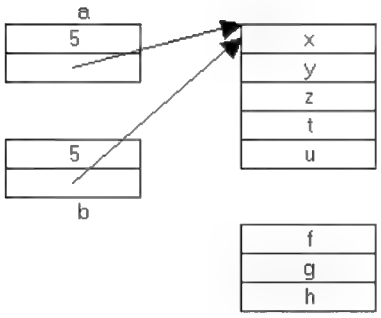
qui correspond au schéma suivant :



L'affectation :

```
b = a ;
```

conduit à :



Le problème est effectivement voisin de celui de la construction par recopie. Voisin, mais non identique, car quelques nuances apparaissent, à savoir :

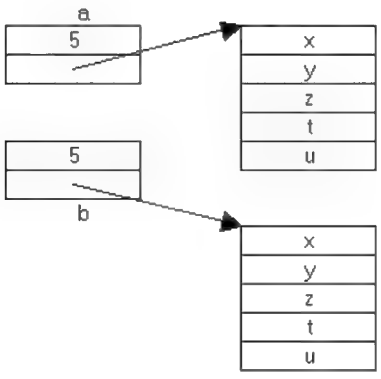
- on peut se trouver en présence d'une affectation d'un objet à lui-même,
- ici, avant affectation, il existe deux objets "complets" (c'est-à-dire avec leur partie dynamique). Dans le cas de la construction par recopie, il n'existait qu'un seul emplacement dynamique, le second étant à créer. On va donc, ici, se retrouver avec l'ancien emplacement dynamique de b ; il n'est plus référencé par b, mais est-on sûr qu'il n'est pas référencé par ailleurs ?

En fait, nous pouvons régler les différents points en surdéfinissant l'opérateur d'affectation, de manière que chaque objet de type vect comporte son propre emplacement dynamique. Dans ce cas, on est sûr qu'il n'est référencé qu'une seule fois et son éventuelle libération peut se faire sans problème. Notez cependant que cette démarche ne convient totalement que si elle est associée à la définition conjointe du constructeur de recopie.

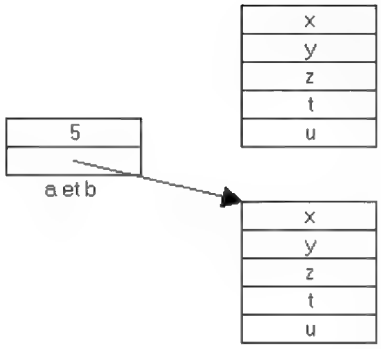
Voici donc comment nous pourrions traiter une affectation telle que b = a, dans le cas (pour l'instant) où a est différent de b :

- libération de l'emplacement pointé par b,
- création dynamique d'un nouvel emplacement dans lequel on recopie les valeurs de l'emplacement pointé par a,
- mise en place des valeurs des membres donnée de b.

Voici un schéma illustrant la situation à laquelle on aboutit :



Par ailleurs, il nous faut prévoir de ne rien faire lorsque a et b désignent le même objet. Dans le cas contraire, l'algorithme proposé ne serait pas satisfaisant. En effet, en supposant que a est transmis par valeur, il conduirait à cette situation :



Celle-ci ne serait pas catastrophique, mais l'ancien emplacement pointé par a ne pourrait pas être libéré. En revanche, si a était transmis par référence, on aboutirait à une double libération d'un même emplacement mémoire.

Enfin, il nous faut "décider" de la valeur de retour fournie par notre opérateur. A ce niveau, tout dépend de l'usage que nous souhaitons en faire :

- si nous nous contentons d'affectations simples (b = a), nous n'avons besoin d'aucune valeur de retour (void),
- si, en revanche, nous souhaitons pouvoir traiter une affectation multiple ou, plus généralement faire en sorte que (comme on peut s'y attendre !) l'expression b = a ait une valeur (probablement celle de b !), il est nécessaire que notre opérateur fournisse une valeur de retour.

Nous choisissons ici la seconde possibilité qui a le mérite d'être plus générale[10]. Voici finalement ce que pourrait être la définition de notre opérateur = (C++ nous impose de le définir comme une fonction membre) : par rapport à nos précédentes explications, b devient le premier opérande 150 – ici this – et a devient le second opérande - ici v. De plus, nous prévoyons de transmettre le second opérande par référence.

```
vect vect::operator = (const vect & v) // notez const
{  if (this != &v)
      { delete adr ;
        adr = new int [nelem = v.nelem] ;
        for (int i=0 ; i<nelem ; i++) adr[i] = v.adr[i] ;
      }
   return * this ;
}
```

Notez que, compte tenu de sa transmission par référence, il est nécessaire d'introduire le qualificatif const pour l'unique argument de notre fonction membre operator=, afin de traiter convenablement le cas de l'affectation d'un vecteur constant à un vecteur quelconque (n'oubliez pas qu'ainsi notre opérateur pourra indifféremment s'appliquer à des objets constants ou non ; dans le cas contraire, il n'aurait pu s'appliquer qu'à des objets non constants).

Nous vous proposons d'intégrer cette définition dans un programme complet servant à illustrer le fonctionnement de l'opérateur. Pour ce faire, nous ajoutons, comme d'habitude, un certain nombre d'instructions d'affichage (en particulier, nous suivons les adresses des objets et des emplacements dynamiques qui leur sont associés). Par contre, pour que le programme ne soit pas trop long, nous avons réduit la classe vect au strict minimum ; en particulier, nous n'y avons pas prévu de **constructeur de recopie** ; or **celui-ci deviendrait naturellement indispensable dans une application réelle.**

Qui plus est, même ici, alors que notre fonction main se limite à l'emploi de l'opérateur =, nous avons dû prévoir une **transmission par référence** pour l'argument et la valeur de retour de operator=. En effet, si

[10] - Bien entendu, C++ vous laisse libre de faire ce que vous voulez, y compris de renvoyer une valeur autre que celle de b (avec tous les risques de manque de lisibilité que cela comporte !).

nous ne l'avions pas fait, l'appel de cet opérateur, traité, rappelons-le, comme une fonction, aurait entraîné un appel de constructeur de recopie (a = b est équivalent ici à : a.operator = (b)) ; il se serait alors agi du constructeur de recopie par défaut, ce qui aurait entraîné les problèmes déjà évoqués de double libération d'un emplacement[11].

```
#include <iostream.h>
class vect
  { int nelem ;                  // nombre d'éléments
    int * adr ;                  // pointeur sur ces éléments
   public :
    vect (int n)                 // constructeur
     { adr = new int [nelem = n] ;
       for (int i=0 ; i<nelem ; i++) adr[i] = 0 ;
       cout << "++ obj taille " << nelem << " en " << this
            << " - v. dyn en " << adr << "\n" ;
     }
    ~vect ()                     // destructeur
     { cout << "-- obj taille " << nelem << " en "
            << this << " - v. dyn en " << adr << "\n" ;
       delete adr ;
     }
    vect & operator = (const vect &) ;    // surdéfinition opérateur =
  } ;

vect & vect::operator = (const vect & v)
{ cout << "== appel opérateur = avec adresses  " << this << " " << &v << "\n" ;
   if (this != &v)
        { cout << "  effacement vecteur dynamique en   " << adr << "\n" ;
          delete adr ;
          adr = new int [nelem = v.nelem] ;
          cout << "  nouveau vecteur dynamique en      " << adr << "\n" ;
          for (int i=0 ; i<nelem ; i++) adr[i] = v.adr[i] ;
        }
      else cout << "  on ne fait rien \n" ;
   return * this ;
}
main()
{   vect a(5), b(3), c(4) ;
    cout << "** affectation a=b \n" ;
    a = b ;
    cout << "** affectation c=c \n" ;
    c = c ;
    cout << "** affectation a=b=c \n" ;
    a = b = c ;
}
```

```
++ obj taille 5 en 0x41140ffa - v. dyn en 0x42c60004
++ obj taille 3 en 0x41140ff4 - v. dyn en 0x42c70004
++ obj taille 4 en 0x41140fee - v. dyn en 0x42c80004
** affectation a=b
```

11 - Un des exercices de ce chapitre vous suggère de le vérifier.

```
== appel opérateur = avec adresses  0x41140ffa 0x41140ff4
  effacement vecteur dynamique en   0x42c60004
  nouveau vecteur dynamique en      0x42c60004
** affectation c=c
== appel opérateur = avec adresses  0x41140fee 0x41140fee
  on ne fait rien
** affectation a=b=c
== appel opérateur = avec adresses  0x41140ff4 0x41140fee
  effacement vecteur dynamique en   0x42c70004
  nouveau vecteur dynamique en      0x42c70004
== appel opérateur = avec adresses  0x41140ffa 0x41140ff4
  effacement vecteur dynamique en   0x42c60004
  nouveau vecteur dynamique en      0x42c60004
-- obj taille 4 en 0x41140fee - v. dyn en 0x42c80004
-- obj taille 4 en 0x41140ff4 - v. dyn en 0x42c70004
-- obj taille 4 en 0x41140ffa - v. dyn en 0x42c60004
```

Exemple d'utilisation d'une classe vect avec un opérateur d'affectation surdéfini

4. NOTION DE FORME CANONIQUE D'UNE CLASSE

Nous avons vu que, dès lors qu'une classe dispose de pointeurs sur des parties dynamiques, la copie d'objets de la classe (aussi bien par le constructeur de recopie par défaut que par l'opérateur d'affectation par défaut) n'est pas satisfaisante. Dans ces conditions, si l'on souhaite que cette recopie fonctionne convenablement, on voit qu'il est nécessaire de munir la classe d' au moins les quatre fonctions membre suivantes :

- constructeur (il sera généralement chargé de l'allocation de certaines parties de l'objet),
- destructeur (il devra libérer correctement tous les emplacements dynamiques créés par l'objet),
- constructeur de recopie,
- opérateur d'affectation.

Voici un canevas récapitulatif correspondant à ce minimum qu'on nomme souvent "classe canonique" :

```
class T
{ public :
   T (...) ;                          // constructeurs autres que par recopie
   T (const T &) ;                    // constructeur de recopie  (forme
conseillée)
   ~T () ;                            // destructeur
   T & T::operator = (const T &) ;    // opérateur d'affectation (forme
conseillée)
   .....
} ;
```

La forme canonique d'une classe

Notez que, bien que ce ne soit pas obligatoire, nous vous conseillons :

- d'employer le qualificatif const pour l'argument du constructeur de recopie et celui de l'affectation, dans la mesure où ces fonctions membre n'ont aucune raison de modifier les valeurs des objets correspondants,
- de prévoir (à moins d'avoir de bonnes raisons de faire le contraire) une valeur de retour à l'opérateur d'affectation, seul moyen de gérer correctement les affectations multiples.

En revanche, l'argument de l'opérateur d'affectation, ainsi que sa valeur de retour peuvent être indifféremment transmis par référence ou par valeur ; cependant, on ne perdra pas de vue que les transmissions par valeur entraînent l'appel du constructeur de recopie ; d'autre part, dès lors que les objets sont de taille respectable, la transmission par référence s'avère plus efficace.

Notez bien que si vous créez une classe comportant des pointeurs, sans la doter de ce "minimum vital" et sans prendre de précautions particulières, l'utilisateur ne se verra nullement interdire la recopie ou l'affectation d'objets.

S'il vous arrive de créer une classe qui n'a pas besoin de disposer des possibilités de copie (ou pour laquelle ces possibilités n'ont pas de sens[12]), plutôt que de compter sur la bonne volonté de l'utilisateur, il est préférable d'utiliser quand même la forme canonique, en s'arrangeant pour interdire certaines actions. Voici quelques pistes dans ce sens :

- si l'opérateur d'affectation d'une classe est surdéfini sous forme privée, toute tentative d'affectation entre objets de cette classe conduira à un message d'erreur en compilation : on dispose donc là d'un moyen d'interdire l'affectation entre objets,
- si le constructeur de recopie est défini sous forme privée, toute tentative d'initialisation (à la construction ou en transmission par valeur) conduira à un message d'erreur en compilation : là encore, on dispose d'un moyen d'interdire la recopie d'objets.

Remarque :

Ce schéma sera complété dans le chapitre XIII afin de prendre en compte la situation d'héritage.

5. EXEMPLE DE SURDÉFINITION DE L'OPÉRATEUR []

Considérons à nouveau notre classe vect :

```
class vect
{     int nelem ;
      int * adr ;
      .....
```

Cherchons à la munir d'outils permettant d'accéder à un élément de l'emplacement pointé par adr, à partir de la connaissance de sa position que l'on repérera par un entier compris entre 0 et nelem-1.

Nous pourrions bien sûr écrire des fonctions membre du genre :

```
void range (int valeur, int position)
```

pour introduire une valeur à une position donnée, et :

12 - C'est généralement le cas des classes "fenêtres" des systèmes graphiques tels que windows...

```
int trouve (int position)
```

pour fournir la valeur située à une position donnée.

La manipulation de nos vecteurs ne serait alors guère aisée. Elle "ressemblerait" à ceci :

```
vect a(5) ;
a.range (15, 0) ;              // place 15 en position 0 de a
a.range (25, 1) ;              // 25 en position 1
for (int i = 2 ; i < 5 ; i++)
    a.range (0, i) ;           // et 0 ailleurs
for i = 0 ; i < 5 ; i++)       // pour afficher les valeurs de a
    cout << a.trouve (i) ;
```

En fait, nous pouvons chercher à surdéfinir l'opérateur [] de manière que a[i] "désigne" l'élément d'emplacement i de a. La seule précaution à prendre consiste à faire en sorte que cette notation puisse être utilisée non seulement dans une expression (cas qui ne présente aucune difficulté), mais également à gauche d'une affectation, c'est-à-dire comme "lvalue". Notez que, dans l'exemple ci-dessus, le problème ne se posait pas, dans la mesure où chaque cas était traité par une fonction membre différente.

Pour que a[i] soit une "lvalue", il est donc nécessaire que la valeur de retour fournie par l'opérateur [] soit transmise par référence.

Par ailleurs, C++ nous impose de surdéfinir cet opérateur sous forme d'une fonction membre, ce qui impose à son premier opérande (notez bien que le premier opérande de a[i] est a) d'être de type classe (ce qui semble raisonnable!). Son prototype sera donc :

```
int & operator [] (int) ;
```

Si nous nous contentons de "renvoyer" l'élément cherché, sans effectuer de contrôle sur la "validité" de la position, le corps de notre fonction operator[] peut se réduire à :

```
return adr[i] ;
```

Voici un exemple simple d'utilisation d'une classe vect, réduite à son strict minimum : constructeur, destructeur et opérateur []. Bien entendu, en pratique, il faudrait au moins lui ajouter un constructeur de recopie et un opérateur d'affectation.

```
#include <iostream.h>
class vect
{
   int nelem ;
   int * adr ;
 public :
   vect (int n) { adr = new int [nelem=n] ; }
   ~vect () {delete adr ;}
   int & operator [] (int) ;
} ;
int & vect::operator [] (int i)
{ return adr[i] ; }
main()
{  int i ;
   vect a(5), b(3), c(3) ;
```

```
    for (i=0 ; i<5 ; i++) {a[i] = i ; b[i] = 2*i ; }
    for (i=0 ; i<3 ; i++) c[i] = a[i]+b[i] ;
    for (i=0 ; i<3 ; i++) cout << c[i] << " " ;
}
```

```
0 3 6
```

Exemple de surdéfinition de l'opérateur[]

Remarques :

1) Nous pourrions bien sûr transmettre le second opérande par référence, mais cela ne présente guère d'intérêt, compte tenu de la petite taille des objets du type vect.

2) Nous pourrions "protéger" la valeur du second opérande contre une modification accidentelle au sein de operator[], en déclarant comme prototype :

```
int & operator [] (const int) ;
```

3) C++ nous interdit de définir l'opérateur [] sous forme d'une fonction amie ; il en allait déjà de même pour l'opérateur =. De toutes façons, nous verrons (dans le prochain chapitre) que, d'une manière générale, il n'est pas conseillé de définir par une fonction amie un opérateur susceptible de modifier un objet, compte tenu des "conversions implicites" pouvant éventuellement être mises en jeu.

4) Tel que nous l'avons conçu, notre opérateur [] ne peut pas être appliqué à un objet constant (puisque seules les fonctions membre dotées du qualificatif const peuvent l'être). Certes, on pourrait se contenter de rajouter ce qualificatif const à notre opérateur mais, dans ce cas, il pourrait modifier un tel objet constant, ce qui n'est pas souhaitable. En général, on préférera définir un second opérateur destiné uniquement aux objets constants en faisant en sorte qu'il puisse "consulter" l'objet en question mais non le modifier. Dans notre cas, voici ce que pourrait être ce second opérateur :

```
int vect::operator [] (int i) const
{ return adr[i] ; }
```

   En effet, une affectation telle que v[i] = ..., v étant un vecteur constant, sera rejetée en compilation puisque notre opérateur transmet son résultat par valeur et non plus par référence.

5) L'opérateur [] était ici dicté par le bon sens, mais nullement imposé par C++. Nous aurions pu tout aussi bien utiliser :

   - l'opérateur () : la notation a(i) aurait encore été compréhensible,
   - l'opérateur < : que penser alors de la notation a < i ?
   - l'opérateur , : notation a, i,
   - etc.

6. SURDEFINITION DE L'OPERATEUR ()

Lorsqu'une classe surdéfinit l'opérateur (), on dit que les objets auxquels elle donne naissance sont des objets fonction car ils peuvent être utilisés de la même manière qu'une fonction. En voici un exemple simple, dans lequel nous surdéfinissons l'opérateur () pour qu'il corresponde à une fonction à deux arguments de type int et renvoyant un int.

```
#include <iostream.h>
class cl_fct
{ public :
   cl_fct(float x) { ..... } ;                  // constructeur
   int operator() (int n, int p ) { ..... }     // opérateur ()
} ;
```

Dans ces conditions, une déclaration telle que :

```
cl_fct obj_fct1(2.5) ;
```

construit bien sûr un objet nommé obj_fct1 de type cl_fct, en transmettant le paramètre 2.5 à son constructeur. En revanche, la notation suivante réalise l'appel de l'opérateur ( ) de l'objet obj_fct1, en lui transmettant les valeurs 3 et 5.

```
obj_fct1(3, 5)
```

Ces possibilités peuvent servir lorsqu'il est nécessaire d'effectuer certaines opérations d'initialisation d'une fonction ou de paramétrer son travail (par le biais des arguments passés à son constructeur). Mais elles s'avéreront encore plus intéressantes dans le cas des fonctions dites de rappel, c'est-à-dire des fonctions qui sont transmises en argument à une autre fonction. Nous en verrons des exemples dans le chapitre XXI consacré aux algorithmes standard.

7. SURDÉFINITION DES OPÉRATEURS NEW ET DELETE

N.B. Ce chapitre peut être ignoré dans un premier temps.

Cette surdéfinition[13], comme celle de n'importe quel autre opérateur, se fait "sélectivement" pour une classe donnée. Bien entendu, il est possible de surdéfinir new et delete pour chacune des classes de votre choix. Les opérateurs prédéfinis[14] continueront d'être employés pour les classes où aucune surdéfinition n'a été réalisée.

7.1 Surdéfinition de new et delete

La surdéfinition de new se fait obligatoirement par une fonction membre. Celle-ci doit :

- recevoir un argument de type size_t ; rappelons que ce type, prévu en C ANSI, est défini dans le fichier en-tête stddef.h (lequel est également inclus par stdlib.h). Cet argument correspond à la taille (en octets) de l'objet à allouer. Bien qu'il figure dans la définition de new, il n'a pas à être spécifié lors de son appel, car c'est le compilateur qui le générera automatiquement (en fonction de la taille de l'objet concerné).
- fournir en retour une valeur de type void * correspondant à l'adresse de l'emplacement alloué pour l'objet.

Quant à la définition de la fonction membre correspondant à l'opérateur delete, elle doit :

- recevoir un argument du type pointeur sur la classe correspondante ; il représente l'adresse de l'emplacement alloué à l'objet à détruire,
- ne fournir aucune valeur de retour (void).

---

13 - Elle n'est possible que depuis la version 2.0.

14 - On dit aussi les opérateurs "globaux".

Remarques

1) Même lorsque l'opérateur new a été surdéfini pour une classe, il reste possible de faire appel à l'opérateur prédéfini en utilisant l'opérateur de résolution de portée ; il en va de même pour delete.

2) Les opérateurs new et delete sont des **fonctions membre statiques** de leur classe (voyez le paragraphe 8 du chapitre VI). En tant que tels, ils n'ont donc accès qu'aux membres statiques de la classe où ils sont définis et ils ne reçoivent pas d'argument implicite (this).

7.2 Exemple

Voici un programme dans lequel la classe point surdéfinit les opérateurs new et delete, dans le seul but d'en comptabiliser les appels[15]. Ils font d'ailleurs appel aux opérateurs prédéfinis (par emploi de ::) pour ce qui concerne la gestion de la mémoire.

```
#include <iostream.h>
#include <stddef.h>                // pour size_t
class point
{   static int npt ;               // nombre total de points
    static int npt_dyn ;           // nombre de points "dynamiques"
    int x, y ;
  public :
    point (int abs=0, int ord=0)                          // constructeur
        { x=abs ; y=ord ;
          npt++ ;
          cout << "++ nombre total de points : " << npt << "\n" ;
        }
    ~point ()                                             // destructeur
        { npt-- ;
          cout << "-- nombre total de points : " << npt << "\n" ;
        }
    void * operator new (size_t sz)                       // new surdéfini
    { npt_dyn++ ;
      cout << "     il y a " << npt_dyn << " points dynamiques sur un \n" ;
      return ::new char[sz] ;
    }
   void operator delete (void * dp)
    { npt_dyn-- ;
      cout << "     il y a " << npt_dyn << " points dynamiques sur un \n" ;
      ::delete (dp) ;
    }
} ;
main()
{  point * ad1, * ad2 ;
   point a(3,5) ;
   ad1 = new point (1,3) ;
   point b ;
   ad2 = new point (2,0) ;
   delete ad1 ;
   point c(2) ;
```

15 - Bien entendu, généralement, dans un programme réel, l'opérateur new aura une tâche plus élaborée.

```
    delete ad2 ;
}

++ nombre total de points : 1
     il y a 1 points dynamiques sur un
++ nombre total de points : 2
++ nombre total de points : 3
     il y a 2 points dynamiques sur un
++ nombre total de points : 4
-- nombre total de points : 3
     il y a 1 points dynamiques sur un
++ nombre total de points : 4
-- nombre total de points : 3
     il y a 0 points dynamiques sur un
-- nombre total de points : 2
-- nombre total de points : 1
-- nombre total de points : 0
```

Exemple de surdéfinition de l'opérateur new pour la classe point

Remarques

1) Comme le montre cet exemple, la surdéfinition des opérateurs new et delete n'a d'incidence que sur les objets alloués dynamiquement. Les objets statiques (alloués à la compilation) et les objets dynamiques (alloués lors de l'exécution, mais sur la pile) ne sont toujours pas concernés.

2) Que new soit surdéfini ou prédéfini, son appel est toujours (heureusement) suivi de celui du constructeur (lorsqu'il existe). De même, que delete soit surdéfini ou prédéfini, son appel est toujours précédé de celui du destructeur (lorsqu'il existe).

3) Lorsqu'ils sont surdéfinis, les opérateurs new et delete ne peuvent pas s'appliquer à des tableaux d'objets. Ainsi, dans l'exemple de programme précédent, une instruction telle que :

```
point * ad = new point [50] ;
```

fera appel (50 fois) à l'opérateur new prédéfini.

4) Il est toujours possible[16] de surdéfinir les opérateurs new et delete de manière globale. Il suffit de définir l'opérateur correspondant sous forme d'une fonction indépendante comme dans cet exemple :

```
void operator new (size_t sz)
{    .....
}
```

Dans ce cas, il faut bien voir que :

- cet opérateur sera appelé pour tous les types pour lesquels aucun opérateur new n'a été surdéfini, **y compris pour les types de base**. Ainsi, la déclaration :

```
int * adi = new int ;
```

y fera appel ;

16 - Dans toutes les versions de C++.

- dans la définition de cet opérateur, il n'est plus possible de faire appel à l'opérateur new prédéfini. Toute tentative d'appel de new ou même de ::new fera entrer dans un processus récursif.

Ce dernier point limite l'intérêt de la surdéfinition globale de new de delete puisque, dans ce cas, le programmeur doit prendre complètement à sa charge la gestion dynamique de mémoire (par exemple en réalisant les "appels au système" nécessaires...).

EXERCICES

N.B. Les exercices marqués (C) sont corrigés en fin de volume.

1) Dans les deux exemples de programmation des paragraphes 1.1 et 1.2, mettez en évidence les appels d'un constructeur de recopie. Pour ce faire, introduisez un constructeur supplémentaire de la forme point (point&) dans la classe point. Voyez ce qui se produit lorsque vous employez, pour le ou les arguments de operator+, la transmission par référence.

2) Dans l'exemple du paragraphe 3, introduisez également un constructeur de recopie pour la classe vect. Constatez que le remplacement de la transmission par référence par une transmission par valeur entraîne la création de nombreux objets supplémentaires.

3 - (C) Dans une classe point de la forme :

```
class point
{     int x, y ;
   public :
      point (int abs = 0, int ord = 0)
            {.....}
      .....
```

introduisez un opérateur ==, tel que si a et b sont deux points, a==b fournisse la valeur 1 lorsque a et b ont les mêmes coordonnées et la valeur 0 dans le cas contraire. On prévoira les deux situations :

a) fonction membre,

b) fonction amie.

4 - (C) Dans une classe pile_entier de la forme[17] :

```
class pile_entier
{     int dim ;           // nombre maxi d'éléments de la pile
      int * adr ;         // adresse du tableau représentant la pile
      int nelem ;         // nombre d'éléments courant de la pile
  public :
      pile_entier (int n) {...}
      ~pile_entier () {...}
      .....
```

introduisez les opérateurs > et <, tels que si p est un objet de type pile_entier et n une variable entière :

17 - Revoyez éventuellement l'exercice VII.5.

p < n ajoute la valeur de n sur la pile p (en ne renvoyant aucune valeur),

p > n supprime la valeur du haut de la pile et la place dans n.

On prévoira les deux situations :

a) fonctions membre,

b) fonctions amies.

5 (C) En langage C, il n'existe pas de véritable type chaîne, mais simplement une "convention" de représentation des chaînes (suite de caractères, terminée par un caractère de code nul). Un certain nombre de fonctions utilisant cette convention permettent les "manipulations" classiques (copie, concaténation...).

Cet exercice vous demande de définir une classe nommée chaine offrant des possibilités plus proches d'un véritable type chaîne (tel que celui du Basic ou du Pascal). Pour ce faire, on prévoira en membres donnée :

- la longueur courante de la chaîne,
- l'adresse d'une zone, allouée dynamiquement, destinée à recevoir la suite de caractères (il ne sera pas nécessaire d'y ranger le caractère nul de fin, puisque la longueur de la chaîne est définie par ailleurs).

Le contenu d'un objet de type chaine pourra donc évoluer facilement par un simple jeu de gestion dynamique.

On munira la classe chaine des constructeurs suivants :

- chaine() : initialise une chaîne vide,
- chaine (char*) : initialise la chaîne avec la chaîne (au sens du C) dont on fournit l'adresse en argument,
- chaine (chaine&) : constructeur de recopie.

On y définira les opérateurs :

- = pour l'affectation entre objets de type chaine (penser à l'affectation multiple),
- == pour examiner l'égalité de deux chaînes,
- + pour réaliser la concaténation de deux chaînes. Si a et b sont de type chaine, a + b sera une (nouvelle) chaîne formée de la concaténation de a et b (les chaînes a et b devront être inchangées).
- [] pour accéder à un caractère de rang donné d'une chaîne (les affectations de la forme a[i] = 'x' devront pouvoir fonctionner.

On pourra ajouter une fonction d'affichage. On ne prévoira pas d'employer de compteur par référence, ce qui signifie qu'on acceptera de dupliquer les chaînes identiques.

N.B. On trouvera dans la bibliothèque standard prévue par la norme, une classe string offrant, entre autres, les fonctionnalités évoquées ici.

X. LES CONVERSIONS DE TYPE DÉFINIES PAR L'UTILISATEUR

N.B. Les possibilités décrites ici sont considérées comme assez dangereuses ; elles doivent n'être employées qu'en toute connaissance de cause. Il est possible d'ignorer ce chapitre dans un premier temps.

En matière de conversion d'un type de base en un autre type de base, C++ offre, naturellement, les mêmes possibilités que le langage C. Rappelons que ces conversions peuvent être explicites ou implicites.

Il s'agit de **conversions explicites** lorsque l'on fait appel à un opérateur de "cast", comme dans :

```
int n ; double z ;
.....
z = double(n) ;
```

ou dans :

```
n = int(z) ;
```

Les **conversions implicites** ne sont, quant à elles, pas mentionnées par "l'utilisateur[1]" et elles sont mises en place par le compilateur, en fonction du contexte. Ces conversions implicites se rencontrent à différents niveaux :

- dans les affectations : il y a alors conversion "forcée" dans le type de la variable réceptrice,
- dans les appels de fonction : comme le prototype est obligatoire en C++, il y a également conversion "forcée" d'un argument dans le type déclaré dans le prototype,
- dans les expressions : dans ce cas, il y a, pour chaque opérateur, conversion éventuelle de l'un des arguments dans le type de l'autre, suivant des règles précises[2]. En particulier, ceci fait intervenir :
  - des conversions systématiques : char et short en int,
  - des conversions d'ajustement de type, par exemple int en long pour une addition de deux valeurs de type long...

Or, C++ permet également de définir des conversions faisant intervenir des types classe créés par l'utilisateur. Par exemple, pour un type complexe, on pourra, moyennant l'écriture de fonctions appropriées, donner une signification aux conversions :

```
complexe -> double
double -> complexe
```

---

1 - C'est-à-dire en fait l'auteur du programme. Nous avons toutefois conservé le terme répandu d'utilisateur ; ce terme s'oppose donc ici à compilateur.

2 - Ces règles sont détaillées dans C-norme ANSI, guide complet de programmation, du même auteur.

Qui plus est, nous verrons que l'existence de telles conversions permettra de donner un sens à l'addition d'un complexe et d'un double, ou même celle d'un complexe et d'un int.

Cependant, s'il paraît à la rigueur logique de disposer de conversions entre une classe complexe et les types numériques, il n'en ira plus nécessairement de même pour des classes n'ayant pas une "connotation" mathématique aussi forte, ce qui n'empêchera pas le compilateur de mettre en place le même genre de conversions !

Ce chapitre fait le point sur ces différentes possibilités. Pour ne pas vous amener trop vite à une conclusion définitive sur leur intérêt ou sur leur absence d'intérêt, nous avons volontairement utilisé des exemples, tantôt signifiants (à connotation mathématique), tantôt non signifiants.

1. LES DIFFÉRENTES SORTES DE CONVERSIONS DÉFINIES PAR L'UTILISATEUR

Considérons une classe point, possédant un **constructeur à un argument**, comme :

```
point (int abs) { x = abs ; y = 0 ; }
```

On peut dire que ce constructeur réalise une conversion d'un int en un objet de type point. Nous avons d'ailleurs déjà vu comment appeler explicitement ce constructeur, par exemple :

```
point a ;
.....
a = point(3) ;
```

Nous verrons que (à moins de l'interdire au moment de la définition de la classe), ce constructeur peut être appelé **implicitement** dans des affectations, des appels de fonction ou des calculs d'expression, au même titre qu'une conversion "usuelle" (on parle aussi de "conversion standard").

D'autre part, nous avons vu que, plus généralement, si l'on considère deux classes nommées point et complexe, un constructeur de la classe complexe à un argument de type point :

```
complexe (point) ;
```

permet de convertir un point en complexe. Là encore, nous verrons que cette conversion peut être utilisée implicitement dans les différentes situations évoquées (à moins qu'on l'ait interdit explicitement).

En revanche, il est facile de voir qu'un constructeur (qui fournit un objet du type de sa classe) ne peut en aucun cas permettre de réaliser une conversion d'un objet en une valeur d'un type de base, par exemple un point en int ou un complexe en double. Comme nous le verrons, ce type de conversion pourra être traité en définissant au sein de la classe concernée, un opérateur de "cast" approprié, par exemple, pour les deux cas cités :

```
operator int()
```

au sein de la classe point,

```
operator double()
```

au sein de la classe complexe.

Cette dernière démarche de définition d'un opérateur de "cast" pourra également être employée pour définir une conversion d'un type classe en un autre type classe. Par exemple, avec :

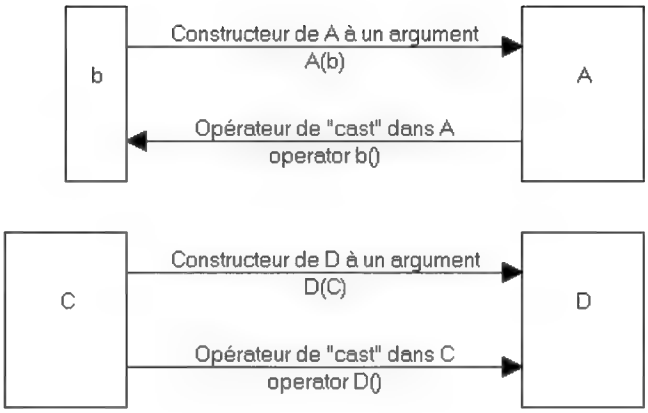
```
operator complexe() ;
```

au sein de la classe point, on définira la conversion d'un point en complexe, au même titre qu'avec le constructeur :

```
complexe (point) ;
```

situé (cette fois) dans la classe complexe.

Voici un schéma récapitulant les différentes possibilités que nous venons d'évoquer : A et B désignent deux classes, b un type de base quelconque.



Parmi les différentes possibilités de conversion que nous venons d'évoquer, seul l'opérateur de "cast", appliqué à une classe, apparaît comme nouveau ; vous pourriez penser que la suite de ce chapitre va se réduire à son étude. C'est effectivement de lui que nous allons maintenant parler. Mais il nous faut également et surtout voir précisément quand et comment les différentes conversions implicites sont mises en œuvre, ainsi que les cas qui sont rejetés par le compilateur.

2. L'OPÉRATEUR DE "CAST" POUR LA CONVERSION D'UN TYPE CLASSE DANS UN TYPE DE BASE

2.1 Définition de l'opérateur de "cast"

Considérons une classe point :

```
class point
{     int x, y ;
      .....
```

Supposez que nous souhaitions la munir d'un opérateur de "cast" permettant la conversion de point en int. Nous le noterons simplement :

```
operator int()
```

Il s'agit là du mécanisme habituel de surdéfinition d'opérateur étudié dans le chapitre précédent : l'opérateur se nomme ici int ; il est unaire (un seul argument) et, comme il s'agit d'une fonction membre, aucun

argument n'apparaît dans son en-tête ou son prototype. Reste la valeur de retour : en principe, cet opérateur fournit un int, de sorte qu'on aurait pu penser à l'en-tête :

```
int operator int()
```

En fait, en C++, **un opérateur de cast doit toujours être défini comme une fonction membre et le type de la valeur de retour** (qui est alors celui défini par le nom de l'opérateur) **ne doit pas être mentionné**.

En définitive, voici comment nous pourrions définir notre opérateur de "cast" (ici "inline"), en supposant que le résultat souhaité pour la conversion en int soit l'abscisse du point.

```
operator int()
{       return x ;
}
```

2.2 Exemple simple d'utilisation

Voici un premier exemple de programme montrant :

- un appel explicite de l'opérateur int que nous venons de définir,
- un appel implicite entraîné par une affectation.

Comme à l'accoutumée, nous avons introduit une instruction d'affichage dans l'opérateur lui-même pour obtenir une trace de son appel.

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point : " << x << " " << y << "\n" ;
     }
   operator int()                        // "cast" point --> int
     { cout << "== appel int() pour le point " << x << " " << y << "\n" ;
       return x ;
     }
} ;
main()
{  point a(3,4), b(5,7) ;
   int n1, n2 ;
   n1 = int(a) ;      // ou  n1 = (int) a      appel explicite de int ()
   cout << "n1 = " << n1 << "\n" ;
   n2 = b ;                                  // appel implicite de int()
   cout << "n2 = " << n2 << "\n" ;
}
```

```
++ construction point : 3 4
++ construction point : 5 7
== appel int() pour le point 3 4
n1 = 3
== appel int() pour le point 5 7
n2 = 5
```

Exemple d'utilisation d'un opérateur de "cast" pour la conversion point -> int

Nous voyons clairement que l'affectation :

```
n2 = b ;
```

a été traduite par le compilateur en :

- une conversion du point b en int,
- une affectation (classique) de la valeur obtenue à n2.

2.3 Appel implicite de l'opérateur de "cast" lors d'un appel de fonction

Nous définissons une fonction fct recevant un argument de type entier. Nous l'appelons :

- une première fois avec un argument entier (6),
- une deuxième fois avec un argument de type point (a).

En outre, nous avons introduit (artificiellement) dans la classe point un constructeur de recopie, ceci afin de montrer qu'ici il n'est pas appelé.

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point : " << x << " " << y << "\n" ;
     }
   point (point & p)                     // constructeur de recopie
     { cout << ":: appel constructeur de recopie \n" ;
       x = p.x ; y = p.y ;
     }
   operator int()                        // "cast" point --> int
     { cout << "== appel int() pour le point " << x << " " << y << "\n" ;
       return x ;
     }
} ;
void fct (int n)                         // fonction
{  cout << "** appel fct avec argument : " << n << "\n" ;
}
main()
{  void fct (int) ;
   point a(3,4) ;
   fct (6) ;                // appel normal de fct
```

```
   fct (a) ;                // appel avec conversion implicite de a en int
}
```

```
++ construction point : 3 4
** appel fct avec argument : 6
== appel int() pour le point 3 4
** appel fct avec argument : 3
```

Appel de l'opérateur de "cast" lors d'un appel de fonction

On voit que l'appel :

```
fct(a)
```

a été traduit par le compilateur en :

- une conversion de a en int,
- un appel de fct, à laquelle on fournit en argument la valeur ainsi obtenue.

Comme on pouvait s'y attendre, la conversion est bien réalisée avant l'appel de la fonction et il n'y a pas de création par recopie d'un objet de type point.

2.4 Appel implicite de l'opérateur de "cast" dans l'évaluation d'une expression

Le programme ci-dessous vous montre comment sont évaluées des expressions telles que a + 3 ou a + b lorsque a et b sont de type point.

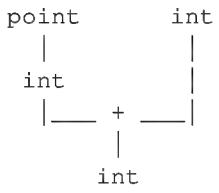
```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point : " << x << " " << y << "\n" ;
     }
   operator int()                        // "cast" point --> int
     { cout << "== appel int() pour le point " << x << " " << y << "\n" ;
       return x ;
     }
} ;
main()
{  point a(3,4), b(5,7) ;
   int n1, n2, n3 ;
   n1 = a + 3 ;    cout << "n1 = " << n1 << "\n" ;
   n2 = a + b ;    cout << "n2 = " << n2 << "\n" ;
```

```
    double z1, z2 ;
    z1 = a + 3 ;    cout << "z1 = " << z1 << "\n" ;
    z2 = a + b ;    cout << "z2 = " << z2 << "\n" ;
}
```

```
++ construction point : 3 4
++ construction point : 5 7
== appel int() pour le point 3 4
n1 = 6
== appel int() pour le point 3 4
== appel int() pour le point 5 7
n2 = 8
== appel int() pour le point 3 4
z1 = 6
== appel int() pour le point 3 4
== appel int() pour le point 5 7
z2 = 8
```

Utilisation de l'opérateur de "cast" dans l'évaluation d'une expression

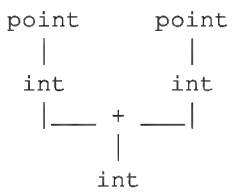
Lorsqu'il rencontre une expression comme a + 3 avec un opérateur portant sur un élément de type point et un entier, le compilateur recherche tout d'abord s'il existe un opérateur + surdéfini correspondant à ces types d'opérandes. Ici, il n'en trouve pas. Il va alors chercher à mettre en place des conversions des opérandes permettant d'aboutir à une opération existante. Dans notre cas, précisément, il va prévoir la conversion de a en int, de manière à se ramener à la somme de deux entiers, suivant le schéma :



Certes, une telle démarche peut choquer. En fait, il faut faire quelques remarques :

- ici, aucune autre conversion n'est envisageable ; il n'en irait pas de même s'il existait un opérateur (surdéfini) d'addition de deux points,
- la démarche paraît moins choquante si l'on ne cherche pas à donner une véritable signification à l'opération a + 3,
- nous cherchons à vous présenter les différentes situations que l'on risque de rencontrer, non pas pour vous encourager à les employer toutes, mais plutôt pour vous mettre en garde.

Quant à l'évaluation de a + b, elle se fait carrément suivant le schéma suivant :



Nous avons prévu, pour chacune des deux expressions évoquées, deux sortes d'affectation :

- à une variable entière,
- à une variable de type double : dans ce cas, il y a conversion forcée du résultat de l'expression en double.

Notez bien que le type de la variable réceptrice n'agit aucunement sur la manière dont l'expression est évaluée pas plus que sur son type final.

2.5 Conversions en chaîne

Considérez cet exemple :

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point : " << x << " " << y << "\n" ;
     }
   operator int()                        // "cast" point --> int
     { cout << "== appel int() pour le point " << x << " " << y << "\n" ;
       return x ;
     }
} ;

void fct (double v)
{  cout << "** appel fct avec argument : " << v << "\n" ;
}


main()
{
   point a(3,4) ;
   int n1 ;
   double z1, z2 ;

   n1 = a + 3.85 ; cout << "n1 = " << n1 << "\n" ;
   z1 = a + 3.85 ; cout << "z1 = " << z1 << "\n" ;
   z2 = a        ; cout << "z2 = " << z2 << "\n" ;
   fct (a) ;
}
```

```
++ construction point : 3 4
== appel int() pour le point 3 4
n1 = 6
== appel int() pour le point 3 4
z1 = 6.85
```

```
== appel int() pour le point 3 4
z2 = 3
== appel int() pour le point 3 4
** appel fct avec argument : 3
```

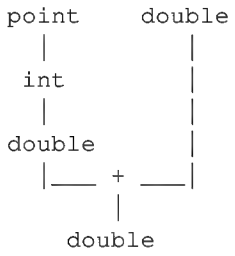
Conversions en chaîne

Cette fois, nous avons à évaluer, à deux reprises, la valeur de l'expression :

```
a + 3.85
```

La différence avec les situations précédentes réside dans le fait que la constante 3.85 est de type double, et non plus de type int. Dans ce cas, on pourrait supposer, par analogie avec ce qui précède, que le compilateur va prévoir la conversion de 3.85 en int. Or, il s'agirait là d'une conversion d'un type de base double en un autre type de base int qui risquerait d'être **"dégradante"** et qui, comme d'habitude, n'est **jamais mise en œuvre de manière implicite dans un calcul d'expression**[3].

En fait, l'évaluation se fera suivant le schéma ci-après :



La valeur affichée pour z1 confirme le type double de l'expression.

Ici, donc, la valeur de a a été soumise à **deux conversions successives** avant d'être transmise à un opérateur. Notez bien que ceci est indépendant de l'usage qui doit être fait ultérieurement de la valeur de l'expression, à savoir :

- conversion en int pour affectation à n1 dans le premier cas,
- affectation à z2 dans le second cas.

Quant à l'affectation z2 = a, elle entraîne une double conversion de point en int, puis de int en double.

Il en va de même pour l'appel :

```
fct (a)
```

D'une manière générale, C++ peut ainsi, en cas de besoin, mettre en œuvre une "chaîne" de conversions, à condition toutefois que celle-ci ne fasse intervenir qu'**une seule C.D.U.** (Conversion Définie par l'Utilisateur[4]). Plus précisément, cette chaîne peut être formée d'au maximum trois conversions, à savoir :

---

[3] - Elle pourrait l'être, bien sûr, dans une affectation ou un appel de fonction, en tant que conversion "forcée".

[4] - Nous avons déjà rencontré ce mécanisme dans le cas des fonctions surdéfinies. Ici, on peut donc dire qu'il s'agit d'un mécanisme comparable appliqué à un opérateur prédéfini, et non plus à une fonction définie par l'utilisateur. Nous retrouverons par la suite des situations semblables, relatives cette fois à un opérateur défini par l'utilisateur (donc à une fonction) ; les règles appliquées seront alors bien celles que nous avons évoquées dans la recherche de la "bonne fonction surdéfinie".

éventuellement une conversion standard, suivie éventuellement d'une C.D.U, suivie éventuellement d'une conversion standard.

2.6 En cas d'ambiguïté

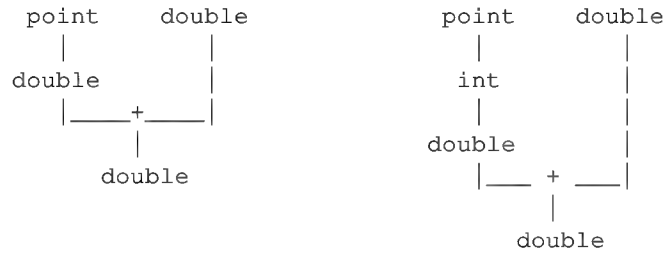
A partir du moment où le compilateur accepte de mettre en place une chaîne de conversions, certaines ambiguïtés peuvent apparaître. Prenons à nouveau l'exemple de notre classe point, en supposant cette fois que nous l'avons munie de deux opérateurs de "cast" :

```
operator int()
operator double()
```

Supposons que nous utilisions à nouveau une expression telle que (a étant de type point) :

```
a + 3.85
```

Dans ce cas, le compilateur se trouve en présence de deux schémas possibles de conversion :



Ici, précisément, il refusera l'expression en fournissant un diagnostic d'ambiguïté.

Cette ambiguïté réside dans le fait que deux chaînes de conversions permettent de passer du type point au type double. S'il s'agissait d'une ambiguïté concernant le choix de l'opérateur à appliquer (ce qui n'était pas le cas ici), le compilateur appliquerait alors les règles habituelles de choix d'une fonction surdéfinie[5].

3. LE CONSTRUCTEUR POUR LA CONVERSION D'UN TYPE DE BASE EN UN TYPE CLASSE

3.1 Un premier exemple

Nous avons déjà vu comment appeler explicitement un constructeur. Par exemple, avec la classe point précédente, si a est de type point, nous pouvons écrire :

```
a = point (12) ;
```

Cette instruction provoque :

- la création d'un objet temporaire de type point,
- l'affectation de cet objet à a.

---

5 - En toute rigueur, il faudrait considérer que les opérateurs sur les types de base correspondent, eux aussi, à des fonctions de la forme operator +.

On peut donc dire que l'expression :

```
point (12)
```

exprime la conversion de l'entier 12 en un point.

D'une manière générale, tout constructeur à un seul argument d'un type de base[6] réalise une conversion de ce type de base dans le type de sa classe.

Or, tout comme l'opérateur de "cast", ce constructeur peut être également appelé implicitement. C'est ainsi que l'affectation :

```
a = 12
```

provoquera exactement la même chose que l'affectation :

```
a = point (12)
```

En effet, à sa rencontre, le compilateur cherchera s'il existe une conversion (voire une chaîne de conversions) unique, permettant de passer du type int au type point. Ici, le constructeur fera l'affaire.

De la même façon, si fct a pour prototype :

```
void fct (point) ;
```

un appel tel que :

```
fct (4)
```

entraînera une conversion de l'entier 4 en un point temporaire qui sera transmis à fct. Notez bien que, dans ce cas, le langage C++ ne précise pas s'il y aura ou non appel du constructeur de recopie[7].

Voici un petit programme illustrant ces premières possibilités de conversion par un constructeur :

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point " << x << " " << y
        << " en " << this << "\n" ;
     }
   point (point & p)                     // constructeur de recopie
     { x = p.x ; y = p.y ;
       cout << ":: constr. recopie de " << &p << " en " << this << "\n" ;
     }
} ;
void fct (point p)                // fonction simple
```

---

6 - Ou, éventuellement, comme c'est le cas ici, à plusieurs arguments ayant des valeurs par défaut, à partir du moment où il peut être appelé avec un seul argument.

7 - Nous avons rencontré les deux possibilités.

```
{  cout << "** appel fct " << "\n" ;
}
main()
{  void fct (point) ;
   point a(3,4) ;
   a = point (12) ;  // appel explicite constructeur
   a = 12 ;          // appel implicite
   fct(4) ;
}
```

```
++ construction point  3 4 en 0x24526265E
++ construction point  12 0 en 0x24522662
++ construction point  12 0 en 0x24522666
++ construction point  4 0 en 0x2453266A
** appel fct
```

Utilisation d'un constructeur pour réaliser des conversions int -> point

Remarque

Bien entendu, si fct est surdéfinie, le choix de la bonne fonction se fera suivant les règles déjà rencontrées dans le chapitre IV. Cette fonction devra être unique et, de plus, les chaînes de conversions mises en œuvre pour chaque argument devront être uniques.

3.2 Le constructeur dans une chaîne de conversions

Supposez que nous disposions d'une classe complexe :

```
class complexe
{     double reel, imag ;
  public :
      complexe (double r = 0 ; double i = 0) ;
      .....
```

Son constructeur permet des conversions double -> complexe. Mais, compte tenu des possibilités de conversion implicite int -> double, ce constructeur peut intervenir dans une chaîne de conversions :

int -> double -> complexe

Ce sera le cas, par exemple, dans une affectation telle que (c étant de type complexe) :

```
c = 3 ;
```

Notez qu'ici cette possibilité de chaîne de conversions rejoint les règles concernant les conversions habituelles à propos des fonctions (surdéfinies ou non). En effet, on peut considérer ici que l'entier 3 est converti en double, compte tenu du prototype de complexe. Cette double interprétation d'une même possibilité n'est pas gênante, dans la mesure où elle conduit, dans les deux cas, à la même conclusion concernant la faisabilité de la conversion.

3.3 Choix entre constructeur ou opérateur d'affectation

Dans l'exemple d'affectation :

```
a = 12
```

du paragraphe 3.1, il n'existait pas d'opérateur d'affectation d'un int à un point. Si tel est le cas, on peut penser qu'alors le compilateur se trouve en présence d'un conflit entre :

- utiliser la conversion int -> point offerte par le constructeur, suivie d'une affectation point -> point,
- utiliser l'opérateur d'affectation int -> point.

En fait, il existe une règle qui permet de trancher : **les conversions définies par l'utilisateur** ("cast" ou constructeur) **ne sont mises en œuvre que lorsque cela est nécessaire.**

C'est donc la seconde solution qui sera choisie ici par le compilateur, comme le montre le programme suivant. Nous y avons surdéfini l'opérateur d'affectation non seulement dans le cas int -> point, mais également dans le cas point -> point afin de bien montrer que cette dernière version n'est pas employée dans l'affectation a = 12.

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point " << x << " " << y
            << " en " << this << "\n" ;
     }
   point & operator = (point & p)        // surd_finition affectation point ->
point
     { x = p.x ; y = p.y ;
       cout << "== affectation point --> point de " << &p << " en " << this ;
       return * this ;
     }
   point & operator = (int n)              // surd_finition affectation int ->
point
     { x = n ; y = 0 ;
       cout << "== affectation int   --> point de " << x << " " << y
            << " en " << this << "\n" ;
       return * this ;
     }
} ;
main()
{  point a(3,4) ;
   a = 12 ;
}
```

```
++ construction point  3 4 en 0x2564
== affectation int   --> point de 12 en 0x2564
```

Les conversions définies par l'utilisateur ne sont mises en œuvre que lorsque cela est nécessaire

3.4 Emploi d'un constructeur pour élargir la signification d'un opérateur

Considérons une classe point, munie d'un constructeur à un argument entier et d'un opérateur d'addition fourni sous forme d'une fonction amie (nous verrons un peu plus loin ce qui se passerait dans le cas d'une fonction membre).

```
class point
{     int x, y ;
  public :
      point (int) ;
      friend point operator + (point, point) ;
      .....
```

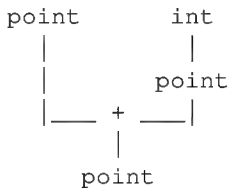
Dans ces conditions, si a est de type point, une expression telle que :

```
a + 3
```

va avoir une signification. En effet, dans ce cas, le compilateur va mettre en œuvre :

- une conversion de l'entier 3 en point (par appel du constructeur),
- l'addition de la valeur obtenue avec celle de a (par appel de operator +).

Le résultat sera du type point. Le schéma suivant récapitule la situation :



On peut dire également que notre expression a + 3 est équivalente (ici) à :

```
operator + (a, point (3))
```

Le même mécanisme s'applique à une expression telle que :

```
5 + a
```

qui sera donc équivalente à :

```
operator + (5, a)
```

Toutefois, dans ce dernier cas, on voit qu'il n'en serait pas allé de même si notre opérateur + avait été défini par une fonction membre. En effet, son premier opérande aurait alors dû être de type point ; aucune conversion implicite n'aurait pu être mise en place dans ce cas[8].

Voici un petit programme illustrant les possibilités que nous venons d'évoquer :

8 - Des appels tels que 5.operator + (a) ou n.operator + (a) (n étant de type int) seront rejetés.

```
#include <iostream.h>
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
     { x = abs ; y = ord ;
       cout << "++ construction point : " << x << " " << y << "\n" ;
     }
   friend point operator + (point, point) ;      // point + point --> point
   void affiche ()
     { cout << "Coordonnées : " << x << " " << y << "\n" ;
     }
} ;
point operator+ (point a, point b)
{  point r ;
   r.x = a.x + b.x ; r.y = a.y + b.y ;
   return r ;
}
main()
{
   point a, b(9,4) ;
   a = b + 5 ; a.affiche() ;
   a = 2 + b ; b.affiche() ;
}
```

```
++ construction point : 0 0
++ construction point : 9 4
++ construction point : 5 0
++ construction point : 0 0
Coordonnées : 14 4
++ construction point : 2 0
++ construction point : 0 0
Coordonnées : 9 4
```

Elargissement de la signification de l'opérateur +

Remarques :

1) La situation évoquée peut rendre de grands services dans une situation réelle puisqu'elle permet de donner un sens à des expressions mixtes. L'exemple le plus caractéristique est celui d'une classe de nombres complexes (supposés constitués de deux valeurs de type double). Il suffit, en effet, de définir la somme de deux complexes et un constructeur à un argument de type double :

```
class complexe
{     double reel, imag ;
  public :
      complexe (double v) { reel = v ; imag = 0 ; }
      friend complexe operator + (complexe, complexe) ;
      .....
```

Les expressions de la forme :

complexe + double

double + complexe

auront alors une signification (et ici ce sera bien celle que l'on souhaite).

Compte tenu des possibilités de conversions, il en ira de même de n'importe quelle addition d'un complexe et d'un float, d'un long, d'un short ou d'un char.

2) Si nous avions défini :

```
class complexe
{     float reel, imag
  public :
      complexe (float v) ;
      friend complexe operator + (complexe, complexe) ;
      .....
```

l'addition d'un complexe et d'un double ne serait pas possible. Elle le deviendrait, par contre, en remplaçant le constructeur par :

```
complexe (double v)
```

(ce qui ne préjuge pas toutefois du résultat de la conversion forcée de double en float qui y figurera !).

3.5 Pour interdire l'utilisation du constructeur dans des conversions implicites : le mot clé explicit

Le projet de norme ANSI de C++ prévoit qu'on puisse interdire l'utilisation du constructeur dans des conversions implicites (simples ou en chaîne), en utilisant le mot clé explicit lors de sa déclaration ; par exemple, avec :

```
class point
{ public :
   explicit point (int) ;
   friend operator + (point, point) ;
     .....
}
```

les instructions suivantes seraient rejetées (a et b étant de type point) :

```
a = 12 ;     // illégal car le constructeur possède le qualificatif explicit
a = b + 5 ;  // idem
```

En revanche, la conversion pourrait toujours se faire par un appel explicite, comme dans :

```
a = point (3) ;      // OK : conversion explicite par le constructeur
a = b + point (5) ;  // idem
```

4. LES CONVERSIONS D'UN TYPE CLASSE EN UN AUTRE TYPE CLASSE

En fait, les possibilités que nous venons de rencontrer, concernant les conversions d'un type de base en un type classe, se généralisent :

- au sein d'une classe A, on peut définir un opérateur de "cast" réalisant la conversion d'un autre type de classe B, dans le type A,
- un constructeur de la classe A, recevant un argument de type B, réalise une conversion de B en A.

4.1 Exemple simple d'opérateur de "cast"

Le programme ci-dessous illustre la première situation : l'opérateur complexe de la classe point permet des conversions d'un objet de type point en un objet de type complexe.

```
#include <iostream.h>
class complexe ;
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) {x=abs ; y=ord ; }
  operator complexe () ;       // conversion point --> complexe
} ;
class complexe
{ double reel, imag ;
 public :
  complexe (double r=0, double i=0) { reel=r ; imag=i ; }
  friend point::operator complexe () ;
  void affiche () { cout << reel << " + " << imag <<"i\n" ; }
} ;
point::operator complexe ()
{ complexe r ; r.reel=x ; r.imag=y ;
  cout << "cast "<<x<<" "<<y<<" en "<<r.reel<<" + "<<r.imag<<"i\n" ;
  return r ;
}
main()
{ point a(2,5) ; complexe c ;
  c = (complexe) a ; c.affiche () ;      // conversion explicite
  point b (9,12) ;
  c =  b ;           c.affiche () ;      // conversion implicite
}
```

```
cast 2 5 en 2 + 5i
2 + 5i
cast 9 12 en 9 + 12i
9 + 12i
```

Exemple d'utilisation d'un opérateur de "cast" pour des conversions point -> complexe

Remarque :

Dans cet exemple, on peut dire que la conversion point -> complexe qui se ramène, au bout du compte, à la conversion de deux entiers en réel est assez naturelle et, de toute façon, pas dégradante. Mais, bien entendu, C++ vous laisse seul juge de la qualité des conversions que vous pouvez définir de cette manière.

4.2 Exemple simple de conversion par un constructeur

Le programme ci-après illustre la deuxième situation : le constructeur complexe (point) représente une autre façon de réaliser des conversions d'un objet de type point en un objet de type complexe.

```
#include <iostream.h>
class point ;
class complexe
{   double reel, imag ;
  public :
    complexe (double r=0, double i=0) { reel=r ; imag=i ; }
    complexe (point) ;
    void affiche () { cout << reel << " + " << imag << "i\n" ; }
} ;
class point
{   int x, y ;
  public :
    point (int abs=0, int ord=0) { x=abs ; y=ord ; }
    friend complexe::complexe (point) ;
} ;
complexe::complexe (point p)
{ reel = p.x ; imag = p.y ; }
main()
{  point a(3,5) ;
   complexe c (a) ; c.affiche () ;
}
```

```
3 + 5i
```

Exemple d'utilisation d'un constructeur pour des conversions point -> complexe

Remarques :

1) La remarque faite précédemment à propos de la "qualité" des conversions s'applique tout aussi bien ici. Nous aurions pu, par exemple, introduire, dans la classe point, un constructeur de la forme point (complexe).

2) En ce qui concerne les conversions d'un type de base en une classe, la seule possibilité qui nous était offerte consistait à prévoir un constructeur approprié au sein de la classe. En revanche, pour les

conversions A -> B (où A et B sont deux classes), nous avons le choix entre placer dans B un constructeur B(A) ou placer dans A un opérateur de "cast" B().

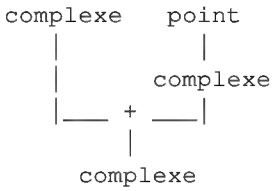
3) Il n'est pas possible de définir simultanément la même conversion A -> B en prévoyant à la fois un constructeur B(A) dans B et un "cast" B() dans A. En effet, ceci conduirait le compilateur à déceler une ambiguïté dès lors qu'une conversion A -> B serait nécessaire. Il faut signaler cependant qu'une telle anomalie peut rester cachée tant que le besoin d'une telle conversion ne se fait pas sentir (en particulier, les classes A et B seront compilées sans problème, y compris si elles figurent dans le même fichier source).

4.3 Pour donner, dans une classe, une signification à un opérateur défini dans une autre classe

Considérons une classe complexe pour laquelle l'opérateur + a été surdéfini par une fonction amie[9], ainsi qu'une classe point munie d'un opérateur de "cast" complexe(). Supposons a de type point, x de type complexe et considérons l'expression :

```
x + a
```

Compte tenu des règles habituelles relatives aux fonctions surdéfinies (mise en œuvre d'une chaîne unique de conversions ne contenant pas plus d'une C.D.U.), le compilateur est conduit à l'évaluation de cette expression suivant le schéma :



Celui-ci fait intervenir l'opérateur + surdéfini par la fonction indépendante operator+. On peut dire que notre expression x + a est en fait équivalente à :

```
operator + (x, a)
```

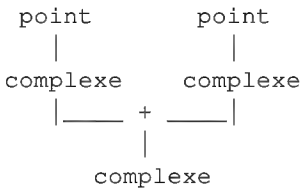
Le même raisonnement s'applique à l'expression a + x. Quant à l'expression :

```
a + b
```

où a et b sont de type point, elle est équivalente à :

```
operator + (a, b)
```

et évaluée suivant le schéma :



9 - Nous verrons que ce point est important : on n'obtiendrait pas les mêmes possibilités avec une fonction membre

Nous vous proposons ci-après un exemple complet de programme illustrant les possibilités que nous venons d'évoquer :

```
#include <iostream.h>
class complexe ;
class point
{   int x, y ;
  public :
    point (int abs=0, int ord=0) { x=abs ; y=ord ; }
    operator complexe () ;
    void affiche () { cout << "point : " << x << " " << y << "\n" ; }
} ;
class complexe
{   double reel, imag ;
  public :
    complexe (double r=0, double i=0) { reel=r ; imag=i ; }
    void affiche () { cout << reel << " + " << imag << "i \n" ; }
    friend point::operator complexe () ;
    friend complexe operator + (complexe, complexe) ;
} ;
point::operator complexe ()
{ complexe r ; r.reel = x ; r.imag = y ; return r ; }
complexe operator + (complexe a, complexe b)
{  complexe r ;
   r.reel = a.reel + b.reel ; r.imag = a.imag + b.imag ;
   return r ;
}

main()
{  point a(3,4), b(7,9), c ;
   complexe  x(3.5,2.8), y ;
   y = x + a ; y.affiche () ;   // marcherait encore si + était fct membre
   y = a + x ; y.affiche () ;   // ne marcherait pas si + était fonction membre
   y = a + b ; y.affiche () ;   // ne marcherait pas si + était fonction membre
                                //       (voir remarque)
                                //  N.B. :  c = a + b n'aurait pas de sens ici
}
```

```
6.5 + 6.8i
6.5 + 6.8i
10 + 13i
```

Elargissement de la signification de l'opérateur + de la classe complexe

Remarques :

1) S'il est effectivement possible ici d'écrire :

```
y = a + b
```

il n'est pas possible d'écrire :

```
c = a + b
```

car il n'existe pas de conversion de complexe(type de l'expression a + b) en point.

Pour que cela soit possible, il suffirait, par exemple, d'introduire dans la classe point un constructeur de la forme point (complexe). Bien entendu, cela ne préjuge nullement de la signification d'une telle opération, et en particulier de son aspect dégradant.

2) Si l'opérateur + de la classe complexe avait été défini par une fonction membre de prototype :

```
complexe complexe::operator + (complexe) ;
```

l'expression a + x n'aurait pas eu de sens, pas plus que a + b. En effet, l'appel de operator +, dans le premier cas, n'aurait pu être que :

```
a.operator + (x)
```

Ceci n'aurait pas été permis. En revanche, l'expression x + a aurait pu correctement être évaluée comme :

```
x.operator + (a)
```

3) Il n'est pas toujours aussi avantageux que dans cet exemple de définir un opérateur sous forme d'une fonction amie. En particulier, si un opérateur modifie son premier opérande (supposé être un objet), il est préférable d'en faire une fonction membre. Dans le cas contraire, en effet, on risque de voir cet opérateur agir tout bêtement, non pas sur l'objet concerné, mais sur un objet (ou une variable) temporaire d'un autre type, créé par une conversion implicite[10]. Notez d'ailleurs que c'est pour cette raison que C++ impose que les opérateurs =, [], () et -> soient toujours surdéfinis par des fonctions membre.

5. QUELQUES CONSEILS

Les possibilités de conversions implicites ne sont certes pas infinies, puisqu'elles sont limitées à une chaîne d'au maximum trois conversions (standard, C.D.U., standard) et que la C.D.U. n'est mise en oeuvre que si elle est utile.

Elles n'en restent pas moins très (trop !) riches ; une telle richesse peut laisser craindre que certaines conversions ne soient mises en place sans que le concepteur des classes concernées ne l'ait souhaité.

En fait, il faut bien voir que :

- l'opérateur de "cast" doit être introduit délibérément par le concepteur de la classe,
- dans la proposition de norme ANSI, le concepteur d'une classe peut interdire l'usage implicite du constructeur dans une conversion, en faisant appel au mot clé explicit.

Avec la norme, il sera donc possible de se protéger totalement contre l'usage des conversions implicites relatives aux classes : il suffira de qualifier tous les constructeurs avec explicit et de ne pas introduire d'opérateur de cast.

D'une manière générale, on aura intérêt à réserver ces possibilités de conversions implicites à des classes ayant une forte "connotation mathématique", dans lesquelles on aura probablement surdéfini un certain nombre d'opérateurs (+, -, etc.).

---

10 - Aucune conversion implicite ne peut avoir lieu sur l'objet appelant une fonction membre

L'exemple le plus classique d'une telle classe est certainement celui de la classe complexe[11] (que nous avons rencontrée dans ce chapitre). Dans ce cas, il paraît naturel de disposer de conversions de complexe en float, de float en complexe, de int en complexe (par le biais de float), etc.

De même, il paraît naturel de pouvoir réaliser aussi bien la somme d'un complexe et d'un float que celle de deux complexes et donc de profiter des possibilités de conversions implicites pour ne définir qu'un seul opérateur d'addition (celle de deux complexes).

11 - Le projet de norme ANSI de C++ contient la définition de classes génériques de complexes.

XI. LES PATRONS DE FONCTIONS

Nous avons déjà vu comment la surdéfinition de fonctions permettait de donner un nom unique à plusieurs fonctions réalisant un travail différent. La notion de "patron[1]" de fonction (on parle aussi de "fonction générique" ou de "modèle de fonction"), introduite par la version 3, est à la fois plus puissante et plus restrictive ; plus puissante car il suffira d'écrire une seule fois la définition d'une fonction pour que le compilateur puisse automatiquement l'adapter à n'importe quel type ; plus restrictive puisque, par nature même, toutes les fonctions ainsi fabriquées par le compilateur devront correspondre à la même définition, donc au même algorithme.

Nous allons commencer par vous présenter cette nouvelle notion sur un exemple simple ne faisant intervenir qu'un seul "paramètre de type". Nous verrons ensuite qu'elle se généralise à un nombre quelconque de tels paramètres et qu'on peut également faire intervenir des "paramètres expression". Puis nous étudierons comment un patron de fonctions peut, à son tour, être surdéfini. Enfin, nous verrons comment toutes ces possibilités peuvent encore être affinées en "spécialisant" une ou plusieurs des fonctions d'un patron.

1. EXEMPLE DE CRÉATION ET D'UTILISATION D'UN PATRON DE FONCTION

1.1 Création d'un patron de fonctions

Supposez que nous ayons besoin d'écrire une fonction fournissant le minimum de deux valeurs de même type reçues en argument. Nous pourrions écrire une définition pour le type int :

```
int min (int a, int b)
{
   if (a < b) return a ;   // ou return a < b ? a : b ;
         else return b ;
}
```

1 - En anglais : template

Bien entendu, il nous faudrait probablement écrire une autre définition pour le type float, c'est-à-dire (en supposant que nous lui donnons le même nom min, ce que nous avons tout intérêt à faire) :

```
float min (float a, float b)
{
   if (a < b) return a ;   // ou return a < b ? a : b ;
         else return b ;
}
```

Nous serions ainsi amené à écrire de nombreuses définitions très proches les unes des autres ; en effet, seul le type concerné serait amené à être modifié.

Depuis la version 3 de C++, nous pouvons simplifier considérablement les choses en définissant **un seul patron de fonctions**, de la manière suivante :

```
#include <iostream.h>
    // création d'un patron de fonctions
template <class T> T min (T a, T b)
{
   if (a < b) return a ;   // ou return a < b ? a : b ;
         else return b ;
}
```

Création d'un patron de fonctions

Comme vous le constatez, seule (ici) l'en-tête de notre fonction a changé (il n'en ira pas toujours ainsi) :

```
template <class T> T min (T a, T b)
```

La mention template < class T> précise que l'on a affaire à un patron (template) dans lequel apparaît un "paramètre[2] de type" nommé T ; notez que C++ a décidé d'employer le mot clé class pour préciser que T est un paramètre de type (on aurait préféré type !). Autrement dit, dans la définition de notre fonction, T représente un type quelconque.

Le reste de l'en-tête :

```
T min (T a, T b)
```

précise que min est une fonction recevant deux arguments de type T et fournissant un résultat du même type.

1.2 Premières utilisations de notre patron de fonctions

Pour utiliser le patron min que nous venons de créer, il suffit tout simplement d'utiliser la fonction min dans des conditions appropriées (c'est-à-dire ici deux arguments de même type). Ainsi, si dans un programme dans lequel n et p sont de type int, nous faisons intervenir l'expression min (n, p), le compilateur

2 - Ou argument ; ici, nous avons convenu d'employer le terme paramètre pour les patrons et le terme argument pour les fonctions ; mais il ne s'agit aucunement d'une convention universelle.

"fabriquera" (on dit aussi "instanciera") automatiquement la fonction min (dite "fonction patron[3]") correspondant à des arguments de type int. Si nous appelons min avec deux arguments de type float, le compilateur "fabriquera" automatiquement une autre fonction patron min correspondant à des arguments de type float et ainsi de suite.

Comme on peut s'y attendre, pour que la chose soit possible, il est nécessaire que le compilateur dispose de la définition du patron en question, autrement dit que les instructions précédentes précèdent une quelconque utilisation de min.

Voici un exemple complet illustrant ceci :

```
#include <iostream.h>
    // création d'un patron de fonctions
template <class T> T min (T a, T b)
{
   if (a < b) return a ;   // ou return a < b ? a : b ;
         else return b ;
}

   // exemple d'utilisation du patron de fonctions min
main()
{
   int n=4, p=12 ;
   float x=2.5, y=3.25 ;
   cout << "min (n, p) = " << min (n, p) << "\n" ;    //  int min(int, int)
   cout << "min (x, y) = " << min (x, y) << "\n" ;    //  float min (float,
loat)
}
```

```
min (n, p) = 4
min (x, y) = 2.5
```

Définition et utilisation d'un patron de fonctions

Remarque :

Les instructions de définition d'un patron ressemblent à des instructions exécutables de définition de fonction. Néanmoins, le mécanisme même des patrons fait que ces instructions sont utilisées par le compilateur pour fabriquer (instancier) chaque fois qu'il est nécessaire les instructions correspondant à la fonction requise ; en ce sens, ce sont donc des déclarations : leur présence est toujours nécessaire et il n'est pas possible de créer un "module objet" correspondant à un patron de fonctions.

En fait, tout se passe comme si, avec la notion de patron de fonctions, apparaissaient deux niveaux de déclarations. On retrouvera le même phénomène pour les patrons de classes. Par la suite, nous continuerons à parler de "définition d'un patron".

En pratique, on placera les définitions de patron dans un fichier approprié d'extension h.

[3] - Attention au vocabulaire : "patron de fonction" pour la fonction générique, "fonction patron" pour une instance donnée.

1.3 D'autres utilisations de notre patron

De façon générale, notre patron min peut être utilisé pour des arguments de *n'importe quel type*, qu'il s'agisse d'un type prédéfini (short, char, double, int *, char *, int * *, etc.) ou d'un type défini par l'utilisateur (notamment structure ou classe).

Par exemple, si n et p sont de type int, un appel tel que min (&n, &p) conduira le compilateur à instancier une fonction int * min (int *, int *).

Examinons ici plus en détail deux situations précises :

- arguments de type char *,
- arguments de type classe.

a) Application au type char *

Voici un premier exemple dans lequel nous exploitons notre patron min pour fabriquer une fonction portant sur des chaînes de caractères :

```
#include <iostream.h>
template <class T> T min (T a, T b)
{
   if (a < b) return a ;   // ou return a < b ? a : b ;
         else return b ;
}
main()
{
   char * adr1 = "monsieur", * adr2 = "bonjour" ;
   cout << "min (adr1, adr2) = " << min (adr1, adr2) ;
}
```

```
min (adr1, adr2) = monsieur
```

Lorsque l'on applique le patron min au type char *

Le résultat peut surprendre, dans la mesure où vous vous attendiez (peut-être) à ce que min fournisse "la chaîne" "bonjour". En fait, il faut bien voir que, à la rencontre de l'expression min (adr1, adr2), le compilateur a généré la fonction suivante :

```
char * min (char * a, char * b)
{
   if (a < b) return a ;
         else return b ;
}
```

La comparaison a< b porte donc sur les valeurs des pointeurs reçus en argment (ici, a était inférieur à b, mais il peut en aller autrement dans d'autres implémentations...). En revanche, l'affichage obtenu par l'opérateur < < porte, non plus sur ces adresses, mais sur les chaînes situées à ces adresses.

b) Application à un type classe

Pour pouvoir appliquer notre patron min à une classe, il est bien sûr nécessaire que l'opérateur < puisse s'appliquer à deux opérandes de ce type classe. En voici un exemple dans lequel nous appliquons min à deux objets de type vect dans lequel l'opérateur < fournit un résultat basé sur le "module" des vecteurs :

```
#include <iostream.h>
    // le patron de fonctions min
template <class T> T min (T a, T b)
{  if (a < b) return a ;
         else return b ;
}
    // la classe vect
class vect
{  int x, y ;
  public :
   vect (int abs=0, int ord=0) { x=abs ; y=ord; }
   void affiche () { cout << x << " " << y ; }
   friend int operator < (vect, vect) ;
} ;
int operator < (vect a, vect b)
{  return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y ;
}
    // un exemple d'utilisation de min
main()
{
   vect u (3, 2), v (4, 1), w ;
   w = min (u, v) ;
   cout << "min (u, v) = " ; w.affiche() ;
}
```

```
min (u, v) = 3 2
```

Utilisation du patron min pour la classe vect

Naturellement, si nous cherchions à appliquer notre patron min à une classe pour laquelle l'opérateur < n'est pas défini, le compilateur le signalerait exactement de la même manière que si nous avions écrit nous-mêmes la fonction min pour ce type.

Remarque :

Un patron de fonctions pourra s'appliquer à des classes patron, c'est-à-dire à un type de classe instancié par un patron de classe. Nous en verrons des exemples dans le prochain chapitre.

2. LES PARAMÈTRES DE TYPE D'UN PATRON DE FONCTIONS

Ce paragraphe fait le point sur la manière dont peuvent intervenir les paramètres de type dans un patron de fonctions, sur l'algorithme qui permet au compilateur d'instancier la fonction voulue et sur les problèmes particuliers qu'il peut poser.

Notez qu'un patron de fonctions peut également comporter ce que l'on nomme des "paramètres expression", lesquels correspondent en fait à la notion usuelle d'argument d'une fonction. Ils seront étudiés dans le paragraphe suivant.

2.1 Utilisation des paramètres de type dans la définition d'un patron

D'une manière générale, un patron de fonctions peut donc comporter un ou plusieurs paramètres de type, chacun devant être précédé du mot clé class, par exemple :

```
template <class T, class U> fct (T a, T * b, U c)
{ ...
}
```

Ces paramètres peuvent intervenir à n'importe quel endroit de la définition d'un patron[4], c'est-à-dire :

- dans l'en-tête (c'était le cas de nos exemples précédents),
- dans des déclarations[5] de variables locales (de l'un des types des paramètres),
- dans les instructions exécutables[6] (par exemple new, sizeof (...)).

En voici un simple exemple d'école :

```
template <class T, class U> fct (T a, T * b, U c)
{
   T x ;                // variable locale x de type T
   U *  adr ;           // variable locale adr de type U *
   ...
   adr = new T [10] ;   // allocation tableau de 10 éléments de type T
   ...
   n = sizeof (T) ;
   ...
}
```

Dans tous les cas, il est nécessaire que **chaque paramètre de type** apparaisse **au moins une fois dans l'en-tête** du patron ; comme nous le verrons, cette condition est parfaitement logique puisque c'est précisément grâce à la nature de ces arguments que le compilateur sera en mesure d'instancier correctement la fonction nécessaire.

---

4 - De la même manière qu'un nom de type peut intervenir dans la définition d'une fonction.

5 - Notez bien qu'il s'agit alors de déclarations, au sein de la définition du patron, c'est-à-dire finalement de déclarations au sein de déclarations.

6 - Nous parlons d'instructions exécutables bien qu'il s'agisse toujours de déclarations (puisque la définition d'un patron est une déclaration) ; en toute rigueur, ces instructions donneront naissance à des instructions exécutables à chaque instanciation d'une nouvelle fonction.

2.2 Identification des paramètres de type d'une fonction patron

Nos précédents exemples étaient suffisamment simples pour que l'on "devine" quelle était la fonction instanciée pour un appel donné. Mais, considérons à nouveau notre patron min :

```
template <class T> T min (T a, T b)
{  if (a < b) return a ;
         else return b ;
}
```

avec ces déclarations :

```
int n ; char c ;
```

Que va faire le compilateur en présence d'un appel tel que min (n,c) ou min(c,n) ? En fait, la règle prévue par C++ dans ce cas est qu'il doit y avoir **correspondance exacte** des types. Cela signifie que nous ne pouvons utiliser notre patron min que pour des appels dans lesquels les deux arguments ont **le même type**. Manifestement, ce n'est pas le cas dans nos deux appels, qui aboutiront à une erreur de compilation. On notera que, dans cette correspondance exacte, les éventuels qualificateurs const ou volatile interviennent.

Voici, à titre indicatif, quelques exemples d'appels de min qui précisent quelle sera la fonction instanciée lorsque l'appel est correct :

```
int n ; char c ; unsigned int q ;
const int ci1 = 10, ci2 = 12 ;
int t[10] ;
int * adi ;
  ...
min (n, c)        // erreur
min (n, q)        // erreur
min (n, ci1)      // erreur : const int et int ne correspondent pas
min (ci1, ci2)    // min (const int, const int)
min (t, adi)      // min (int *, int *) car ici, t est converti en int *, avant
appel
```

Il est toutefois possible, dans un appel de fonction patron, de spécifier tout ou partie des paramètres de type à utiliser. Voici quelques exemples utilisant les déclarations précédentes :

```
min<int> (c, n)      /* force l'utilisation de min<int>, et donc la conversion
*/
                     /* de c en int ; le résultat sera de type int
*/
min<char> (q, n)     /* force l'utilisation de min<char>, et donc la conversion
*/
                     /* de q et de n en char ; le résultat sera de type char
*/
```

Voici un autre exemple faisant intervenir plusieurs paramètres de type :

```
template <class T, class U> T fct (T x, U y, T z)
{  return x + y + z ;
}

main ()
{  int n = 1, p = 2, q = 3 ;
   float x = 2.5, y = 5.0 ;
```

```
    cout << fct (n, x, p) << "\n" ;      // affiche la valeur (int) 5
    cout << fct (x, n, y) << "\n" ;      // affiche la valeur (float) 8.5
    cout << fct (n, p, q) << "\n" ;      // affiche la valeur (int) 6
    cout << fct (n, p, x) << "\n"  ;     // erreur : pas de correspondance
}
```

Ici, encore, on peut forcer certains des paramètres de type, comme dans ces exemples :

```
fct<int,float> (n, p, x)  // force l'utilisation de fct<int,float> et donc la
                          // conversion de p en float et de x en int
fct<float> (n, p, x )     // force l'utilisation de float pour T ; U est
déterminé
                          // par les règles habituelles, c'est-à-dire int (type
de p)
                          // n sera converti en float
```

2.3 Nouvelle syntaxe d'initialisation des variables des types standard

Dès lors que, dans un patron de fonctions, un paramètre de type est susceptible de correspondre tantôt à un type standard, tantôt à un type classe, un problème apparaît si l'on doit déclarer, au sein du patron, un objet de ce type en transmettant un ou plusieurs arguments à son constructeur. Voyez cet exemple :

```
template <class T> fct (T a)
{  T x (3) ;   // x est un objet local de type x qu'on construit
               //  en transmettant la valeur 3 à son constructeur
   // ...
}
```

Tant que l'on utilise une fonction fct pour un type classe, tout va bien. En revanche, si l'on cherche à l'utiliser pour un type standard, par exemple int, le compilateur est amené à générer la fonction suivante :

```
fct (int a)
{  int x (3) ;
   // ...
}
```

Pour que l'instruction int x(3) ne pose pas de problème, C++ a prévu qu'elle soit simplement interprétée comme une initialisation de x avec la valeur 3, c'est-à-dire comme :

```
int x = 3 ;
```

En théorie, cette possibilité est utilisable dans n'importe quelle instruction C++, de sorte que vous pourriez très bien écrire :

```
double x(3.5) ;       // au lieu de      double x = 3.5 ;
char c('e') ;         // au lieu de      char c = 'e' ;
```

En pratique, cela sera rarement utilisé de cette façon.

2.4 Limitations des patrons de fonctions

Lorsque l'on définit un patron de classe, à un paramètre de type peut théoriquement correspondre n'importe quel type effectif (standard ou classe). Il n'existe a priori aucun mécanisme intrinsèque permettant d'interdire l'instanciation pour certains types.

Ainsi, si un patron a un en-tête de la forme :

```
template <class T> void fct (T)
```

on pourra appeler fct avec un argument de n'importe quel type : int, float, int *, int * * t, t * ou même t * * (t désignant un type classe quelconque)...

Cependant, un certain nombre d'éléments peuvent intervenir indirectement pour faire échouer l'instanciation.

Tout d'abord, on peut imposer qu'un paramètre de type corresponde à un pointeur. Ainsi, avec un patron d'en-tête :

```
template <class T> void fct (T *)
```

on ne pourra appeler fct qu'avec un pointeur sur un type quelconque : int *, int * *, t * ou t * *. Dans les autres cas, on aboutira à une erreur de compilation.

Par ailleurs, dans la définition d'un patron peuvent apparaître des instructions qui s'avéreront incorrectes lors de la tentative d'instanciation pour certains types.

Par exemple, notre patron min :

```
template <class T> T min (T a, T b)
{  if (a < b) return a ;
          else return b ;
}
```

ne pourra pas s'appliquer si T correspond à un type classe dans lequel l'opérateur < n'a pas été surdéfini.

De même, un patron de ce genre :

```
template <class T> void fct (T)
{  ...
  T x (2, 5) ; // objet local de type T, initialisé par un constructeur à 2
arguments
   ...
}
```

ne pourra pas s'appliquer à un type classe pour lequel n'existe pas un constructeur à deux arguments.

En définitive, bien qu'il n'existe pas de mécanisme formel de limitation, les patrons de fonctions peuvent néanmoins comporter dans leur définition même un certain nombre d'éléments qui en limiteront la portée.

3. LES PARAMÈTRES EXPRESSION D'UN PATRON DE FONCTIONS

Comme nous l'avons déjà évoqué, un patron de fonctions peut comporter des "paramètres expression". Il s'agit en fait tout simplement de paramètres (muets) "ordinaires", analogues à ceux qu'on trouve dans la

définition d'une fonction. Voyez cet exemple dans lequel nous définissons un patron nommé compte permettant de fabriquer des fonctions comptabilisant le nombre d'éléments nuls d'un tableau de type quelconque et de taille quelconque.

```
#include <iostream.h>
template <class T> int compte (T * tab, int n)
{  int i, nz=0 ;
   for (i=0 ; i<n ; i++) if (!tab[i]) nz++ ;
   return nz ;
}
main ()
{ int t [5] = { 5, 2, 0, 2, 0} ;
  char c[6] = { 0, 12, 0, 0, 0} ;
  cout << "compte (t) = " << compte (t, 5) << "\n" ;
  cout << "compte (c) = " << compte (c, 6) << "\n" ;
}

compte (t) = 2
compte (c) = 4
```

Exemple de patron de fonctions comportant un paramètre expression (n)

On peut dire que notre patron compte définit une famille de fonctions compte, dans laquelle le type du premier argument est variable (et donc défini par l'appel), tandis que le second est de type imposé (ici int). Comme on peut s'y attendre, dans un appel de compte, seul le type du premier argument intervient dans le code de la fonction instanciée.

D'une manière générale un patron de fonctions peut disposer d'un ou de plusieurs paramètres expression. Lors de l'appel, leur type n'a plus besoin de correspondre exactement à celui attendu : il suffit qu'il soit acceptable par affectation, comme dans n'importe quel appel d'une fonction ordinaire.

4. SURDÉFINITION DE PATRONS

De même qu'il est possible de surdéfinir une fonction classique, il est possible de surdéfinir un patron de fonctions, c'est-à-dire de définir plusieurs patrons possédant des arguments différents. On notera que cette situation conduit en fait à définir plusieurs "familles" de fonctions (il y a bien plusieurs définitions de familles, et non plus simplement plusieurs définitions de fonctions) ; elle ne doit pas être confondue avec la spécialisation d'un patron de fonctions qui consiste à surdéfinir une ou plusieurs des fonctions de la famille et que nous étudierons dans le paragraphe suivant.

4.1 Exemples de surdéfinition de patron de fonctions ne comportant que des paramètres de type

Voyez cet exemple, dans lequel nous avons surdéfini deux patrons de fonctions min, de façon à disposer :

- d'une première famille de fonctions à deux arguments de même type quelconque (comme dans nos précédents exemples),
- d'une seconde famille de fonctions à trois arguments de même type quelconque.

```
#include <iostream.h>
   // patron numero I
template <class T> T min (T a, T b)
{
   if (a < b) return a ;
         else return b ;
}
   // patron numero II
template <class T> T min (T a, T b, T c)
{
   return min (min (a, b), c) ;
}
main()
{
  int n=12, p=15, q=2 ;
  float x=3.5, y=4.25, z=0.25 ;
  cout << min (n, p) << "\n" ;      // patron no I    int min (int, int)
  cout << min (n, p, q) << "\n" ;   // patron no II   int min (int, int, int)
  cout << min (x, y, z) << "\n" ;   // patron no II   float min (float, float,
float)
}
```

Exemple de surdéfinition de patron de fonctions (1)

D'une manière générale, on peut surdéfinir des patrons possédant un nombre différent de paramètres de type (dans notre exemple, il n'y en avait qu'un dans chaque patron min) et les en-têtes des fonctions correspondantes peuvent être aussi variés qu'on le désire. Mais il est souhaitable qu'il n'y ait aucun recoupement entre les différentes familles de fonctions correspondant à chaque patron. Si tel n'est pas le cas, une ambiguïté risque d'apparaître avec certains appels.

Voici un autre exemple dans lequel nous avons défini plusieurs patrons de fonctions min à deux arguments, afin de traiter convenablement les trois situations suivantes :

- deux valeurs de même type (comme dans les paragraphes précédents),
- un pointeur sur une valeur d'un type donné et une valeur de ce même type,
- une valeur d'un type donné et un pointeur sur une valeur de ce même type.

```
#include <iostream.h>
   // patron numéro I
```

```
template <class T> T min (T a, T b)
{  if (a < b) return a ;
         else return b ;
}
   // patron numéro II
template <class T> T min (T * a, T b)
{  if (*a < b) return *a ;
          else return b ;
}
   // patron numéro III
template <class T> T min (T a, T * b)
{  if (a < *b) return a ;
          else return *b ;
}
main()
{ int n=12, p=15 ;
  float x=2.5, y=5.2 ;
  cout << min (n, p) << "\n" ;   // patron numéro I    int min (int, int)
  cout << min (&n, p) << "\n" ;  // patron numéro II   int min (int *, int)
  cout << min (x, &y) <<"\n" ;   // patron numéro III  float min (float, float
*)
  cout << min (&n, &p) << "\n" ; // patron numéro I    int * min (int *, int *)
}
```

```
12
12
2.5
0x210d2336
```

Exemple de surdéfinition de patron de fonctions (2)

Les trois premiers appels ne posent pas de problème. En revanche, un appel tel que min (&n, &p) conduit à instancier, à l'aide du patron numéro I la fonction :

```
int * min (int *, int *)
```

La valeur fournie alors par l'appel en question est la plus petite des deux valeurs (de type int *) &n et &p. Il est probable que ce ne soit pas le résultat attendu par l'utilisateur (nous avons déjà rencontré ce genre de problème dans le paragraphe 1 en appliquant min à des chaînes[7]).

Pour l'instant, notez qu'il ne faut pas espérer améliorer la situation en définissant un patron supplémentaire de la forme :

```
template <class T> T min (T * a, T * b)
{  if (*a < *b) return *a ;
           else return *b ;
}
```

7 - Mais ce problème pourra se régler convenablement avec la spécialisation de patron, ce qui n'est pas le cas du problème que nous exposons ici.

En effet, les quatre familles de fonctions ne seraient alors plus totalement indépendantes. Plus précisément, si les trois premiers appels fonctionnent toujours convenablement, l'appel min (&n, &p) conduit alors à une ambiguïté puisque deux patrons conviennent maintenant (celui que nous venons d'introduire et le premier).

4.2 Exemples de surdéfinition de patron de fonctions comportant des paramètres expression

Rappelons que certaines implémentations autorisent les paramètres expression. Dans ces conditions, la surdéfinition de patron prend un caractère plus général. Dans l'exemple suivant, nous avons défini deux familles de fonctions min :

- l'une pour déterminer le minimum de deux valeurs de même type quelconque,
- l'autre pour déterminer le minimum des valeurs d'un tableau de type quelconque et de taille quelconque (fournie en argument sous forme d'un entier).

```
#include <iostream.h>
  // patron I
template <class T> T min (T a, T b)
{  if (a < b) return a ;
         else return b ;
}
  // patron II
template <class T> T min (T * t, int n)
{  int i ;
   T min = t[0] ;
   for (i=1 ; i<n ; i++) if (t[i] < min) min=t[i] ;
   return min ;
}
main()
{  long n=2, p=12 ;
   float t[6] = {2.5, 3.2, 1.5, 3.8, 1.1, 2.8} ;
   cout << min (n, p) ;       // patron I    long min (long, lon)
   cout << min (t, 6) ;       // patron II   float min (float *, int)
}
```

Exemple de surdéfinition de patrons comportant un paramètre expression

Notez que si plusieurs patrons sont susceptibles d'être employés et qu'ils ne se distinguent que par le type de leurs paramètres expression, ce sont alors les règles de choix d'une fonction surdéfinie ordinaire qui s'appliquent.

5. SPÉCIALISATION DE FONCTIONS DE PATRON

Un patron de fonctions définit une famille de fonctions à partir d'une seule définition. Autrement dit, toutes les fonctions de la famille réalisent le même algorithme. Dans certains cas, ceci peut s'avérer pénalisant.

Nous l'avons d'ailleurs déjà remarqué dans le cas du patron min du paragraphe 1 : le comportement obtenu lorsqu'on l'appliquait au type char * ne nous satisfaisait pas.

La notion de spécialisation offre une solution à ce problème. En effet, C++ vous autorise à fournir, outre la définition d'un patron, la définition d'une ou de plusieurs fonctions pour certains types d'arguments. Voici, par exemple, comment améliorer notre patron min du paragraphe 1 en fournissant une version spécialisée pour les chaînes :

```
#include <iostream.h>
#include <string.h>        // pour strcmp
   // patron min
template <class T> T min (T a, T b)
{  if (a < b) return a ; else return b ;
}
   // fonction min pour les chaines
char * min (char * cha, char * chb)
{  if (strcmp (cha, chb) < 0) return cha ;
                         else return chb ;
}
main()
{ int n=12, p=15 ;
  char * adr1 = "monsieur", * adr2 = "bonjour" ;
  cout << min (n, p) << "\n" ;     // patron   int min (int, int)
  cout << min (adr1, adr2) ;       // fonction char * min (char *, char *)
}
```

```
12
bonjour
```

Exemple de spécialisation d'une fonction d'un patron

Remarque

En théorie, d'après la norme ANSI (mais pas dans la version 3), il est possible d'effectuer ce que l'on nomme des spécialisations partielles, c'est-à-dire de définir des familles de fonctions, certaines étant plus générales que d'autres, comme dans :

```
template <class T, class U> void fct (T a, U b) { ..... }
template <class T>          void fct (T a, T b) { ..... }
```

Manifestement la deuxième définition est plus spécialisée que la première et devrait être utilisée dans des appels de fct dans lesquels les deux arguments sont de même type.

En pratique, toutes les implémentations ne traitent pas encore convenablement de telles situations et nous n'entrerons pas plus dans les détails.

6. LES PATRONS DE FONCTIONS D'UNE MANIÈRE GÉNÉRALE

D'une manière générale, on peut donc définir un ou plusieurs patrons de même nom (surdéfinition), chacun possédant ses propres paramètres de type et éventuellement des paramètres expression. De plus, il est possible de fournir des fonctions "ordinaires" portant le même nom qu'un patron ; on parle dans ce cas de spécialisation d'une fonction de patron.

Ce paragraphe se propose de faire le point concernant l'algorithme utilisé par le compilateur dans l'instanciation (ou l'appel) de la fonction correspondant à un appel donné.

Dans un premier temps, on examine toutes les fonctions "ordinaires" ayant le nom voulu et on s'intéresse aux correspondances exactes. Si une seule convient, le problème est résolu. S'il en existe plusieurs, il y a ambiguïté ; une erreur de compilation est détectée et la recherche est interrompue.

Si aucune fonction ordinaire ne réalise de correspondance exacte, on examine alors tous les patrons ayant le nom voulu, en ne considérant que les paramètres de type. Si une seule correspondance exacte est trouvée, la fonction correspondante est instanciée[8] et le problème est résolu. S'il y en a plusieurs, on examine tout d'abord si l'on est en présence d'une spécialisation partielle, auquel cas l'on choisit le patron le plus spécialisé[9] ; sinon l'ambiguïté conduit à une erreur de compilation et la recherche est interrompue

Enfin, si aucun patron de fonctions ne convient, on examine à nouveau toutes les fonctions "ordinaires" en les traitant cette fois comme de simples fonctions surdéfinies (promotions numériques, conversions standard[10]...).

Remarque

Il est tout à fait possible que la définition d'un patron fasse intervenir à son tour une fonction patron (c'est-à-dire une fonction susceptible d'être instanciée à partir d'un autre patron).

---

8 - Du moins, si elle n'a pas déjà été instanciée.

9 - Rappelons que la possibilité de spécialisation partielle des patrons de fonctions n'est pas correctement gérée par toutes les implémentations.

10 - Revoyez éventuellement le paragraphe 5.3 du chapitre 4.

XII. LES PATRONS DE CLASSES

Nous avons vu dans le précédent chapitre comment C++ permettait, grâce à la notion de patron de fonctions, de définir une famille de fonctions paramétrées par un ou plusieurs types et, éventuellement, des expressions. D'une manière comparable, C++ vous permet (également depuis la version 3) de définir des "patrons de classes". Là encore, il suffira décrire une seule fois la définition de la classe pour que le compilateur puisse automatiquement l'adapter à différents types.

Comme nous l'avons fait pour les patrons de fonctions, nous commencerons par vous présenter cette notion de patron de classes sur un exemple simple ne faisant intervenir qu'un paramètre de type. Nous verrons ensuite qu'elle se généralise à un nombre quelconque de paramètres de type et de paramètres expression. Puis nous examinerons la possibilité de spécialiser un patron de classes, soit en spécialisant certaines de ses fonctions membre, soit en spécialisant toute une classe. Nous ferons alors le point sur l'instanciation de classes patron, notamment en ce qui concerne l'identité de deux classes. Nous verrons ensuite comment se généralisent les déclarations d'amitiés dans le cas de patrons de classes. Nous terminerons sur un exemple d'utilisation de classes patrons imbriquées en vue de manipuler des tableaux (d'objets) à deux indices.

Signalons dès maintenant que, malgré leurs ressemblances, les notions de patron de fonctions et de patron de classes recèlent des différences assez importantes. Comme vous le verrez, ce chapitre n'est nullement l'extrapolation aux classes du précédent chapitre consacré aux fonctions.

1. EXEMPLE DE CRÉATION ET D'UTILISATION D'UN PATRON DE CLASSES

1.1 Création d'un patron de classes

Nous avons souvent été amené à créer une classe point de ce genre (nous ne fournissons pas ici la définition des fonctions membre) :

```
class point
{  int x ; int y ;
  public :
   point (int abs=0, int ord=0) ;
   void affiche () ;
   // .....
}
```

Lorsque nous procédons ainsi, nous imposons aux coordonnées d'un point d'être des valeurs de type int. Si nous souhaitons disposer de points à coordonnées d'un autre type (float, double, long, unsigned int...), nous devons définir une autre classe en remplaçant simplement, dans notre précédente classe, le mot clé int par le nom de type voulu.

Nous pouvons, ici encore, simplifier considérablement les choses en définissant un seul patron de classe en procédant ainsi :

```
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0) ;
   void affiche () ;
} ;
```

Comme dans le cas des patrons de fonctions, la mention template < class T> précise que l'on a affaire à un patron (template) dans lequel apparaît un paramètre de type nommé T ; rappelons que C++ a décidé d'employer le mot clé class pour préciser que T est un argument de type (pas forcément classe...).

Bien entendu, la définition de notre patron de classes n'est pas encore complète puisqu'il y manque la définition des fonctions membre, à savoir, ici, le contructeur point et la fonction affiche. Pour ce faire, la démarche va légèrement différer suivant que la fonction concernée est en ligne ou non.

Pour une fonction en ligne, les choses restent naturelles ; il suffit simplement d'utiliser le paramètre T à bon escient. Voici par exemple comment pourrait être défini notre constructeur :

```
point (T abs=0, T ord=0)
{   x = abs ; y = ord ;
}
```

En revanche, lorsque la fonction est définie en dehors de la définition de la classe, il est nécessaire de rappeler au compilateur :

- que, dans la définition de cette fonction, vont apparaître des paramètres de type ; pour ce faire, on fournira à nouveau la liste de paramètre sous la forme :

```
template <class T>
```

- le nom du patron concerné (de même qu'avec une classe "ordinaire", il fallait préfixer le nom de la fonction du nom de la classe...) ; par exemple, si nous définissons ainsi la fonction affiche, son nom sera :

```
point<T>::affiche ()
```

En définitive, voici comment se présenterait l'en-tête de la fonction affiche si nous le définissions ainsi en dehors de la classe :

```
template <class T> void point<T>::affiche ()
```

Notez qu'en toute rigueur le rappel du paramètre T à la suite du nom de patron (point) est "redondant[1]" puisque ce paramètre a déjà été spécifié dans la liste de paramètres suivant le mot clé template.

Voici ce que pourrait être finalement la définition de notre patron point :

```
#include <iostream.h>
  // création d'un patron de classe
template <class T> class point
```

1 - Stroustrup, lui-même, se contente de mentionner cette redondance, sans la justifier.

```
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)
   {  x = abs ; y = ord ;
   }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{
  cout << "Coordonnées : " << x << " " << y << "\n" ;
}
```

Création d'un patron de classes

1.2 Utilisation d'un patron de classes

Une fois ainsi créé un tel patron, une déclaration telle que :

```
point <int> ai ;
```

amènera le compilateur à "instancier" la définition d'une classe point dans laquelle le paramètre T prend la valeur int. Autrement dit, tout se passera comme si nous avions fourni une définition complète de cette classe.

Si nous déclarons :

```
point <double> ad ;
```

le compilateur instanciera la définition d'une classe point dans laquelle le paramètre T prend la valeur double, exactement comme si nous avions fourni une autre définition complète de cette classe.

Si nous avons besoin de fournir des arguments au constructeur, nous le ferons classiquement comme dans :

```
point <int> ai (3, 5) ;
point <double> ad (3.5, 2.3) ;
```

Remarque :

Comme on peut s'y attendre, les instructions définissant un patron de classes sont des déclarations au même titre que le sont les instructions définissant une classe (y compris les instructions de définition de fonctions en ligne).

Mais il en va de même pour les fonctions membre qui ne sont pas en ligne : leurs instructions sont nécessaires au compilateur pour instancier chaque fois que nécessaire les instructions requises. On retrouve ici la même remarque que celle que nous avons formulée pour les patrons de fonctions.

Ainsi n'est-il pas possible de livrer à un utilisateur une classe patron toute compilée : il faudra lui en fournir les instructions source de toutes les fonctions membre (alors que pour une classe "ordinaire" il suffit de lui fournir la déclaration de la classe et un module objet correspondant aux fonctions membre).

1.3 Exemple récapitulatif

Voici un exemple complet reprenant à la fois :

- la création d'un patron de classes point, comportant un constructeur en ligne et une fonction membre (affiche) non en ligne,
- un exemple d'utilisation.

```
#include <iostream.h>
  // création d'un patron de classe
template <class T> class point
{
   T x ; T y ;
  public :
   point (T abs=0, T ord=0)
   {  x = abs ; y = ord ;
   }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{
  cout << "Coordonnées : " << x << " " << y << "\n" ;
}

main ()
{
   point <int> ai (3, 5) ;         ai.affiche () ;
   point <char> ac ('d', 'y') ;    ac.affiche () ;
   point <double> ad (3.5, 2.3) ; ad.affiche () ;
}
```

```
coordonnées : 3 5
coordonnées : d y
coordonnées : 3.5 2.3
```

Création et utilisation d'un patron de classes

Remarques :

1) Le comportement de point< char> est satisfaisant si nous souhaitons effectivement disposer de points repérés par des (vrais) caractères. Si, en revanche, nous avons utilisé le type char pour disposer de "petits entiers", le résultat est moins satisfaisant ; en effet, nous pourrons certes toujours déclarer un point de cette façon :

```
point <char> pc (4, 9) ;
```

Mais le comportement de la fonction affiche ne nous conviendra plus (nous obtiendrons les caractères ayant pour code les coordonnées du point !).

Nous verrons qu'il reste toujours possible de modifier cela en "spécialisant" notre classe point pour le type char ou encore en spécialisant la fonction affiche pour la classe point< char> .

2) A priori, on a plutôt envie d'appliquer notre patron point à des types T standard. Toutefois, rien n'interdit de l'appliquer à un type classe T quelconque. Hormis le fait que, dans ce cas, il peut être difficile d'attribuer une signification à la classe patron ainsi obtenue, il est nécessaire qu'il existe alors une conversion de int en T (utile pour convertir la valeur 0 dans le type T lors de l'initialisation des arguments du constructeur de point). De plus, il est nécessaire que la recopie et l'affectation d'objets de type T soient correctement prises en compte.

2. LES PARAMÈTRES DE TYPE D'UN PATRON DE CLASSES

Tout comme les patrons de fonctions, les patrons de classes peuvent comporter des paramètres de type et des paramètres expression. Ce paragraphe étudie les premiers ; les seconds seront étudiés dans le paragraphe suivant. Une fois de plus, notez bien que, malgré leur ressemblance avec les patrons de fonctions, les contraintes relatives à ces différents types de paramètres ne seront pas les mêmes.

2.1 Les paramètres de type dans la création d'un patron de classes

Les paramètres de type peuvent être en nombre quelconque et ils sont utilisés comme bon vous semble dans la définition du patron de classes. En voici un exemple :

```
template <class T, class U, class V> // liste de trois param. de nom (muet) T,
U et V
class essai
{  T x ;              // un membre x de type T
   U t[5]       ;     // un tableau t de 5 éléments de type U
     ...
   V fm1 (int, U) ;   // déclaration d'une fonction membre recevant 2 arguments
                      //  de type int et U et renvoyant un résultat de type V
     ...
} ;
```

2.2 Instanciation d'une classe patron

Rappelons que nous nommons "classe patron" une instance particulière d'un patron de classe.

Une classe patron se déclare simplement en fournissant à la suite du nom de patron un nombre d'arguments effectifs (noms de types) égal au nombre de paramètres figurant dans la liste (template < ...>) du patron. Voici des déclarations de classes patron obtenues à partir du patron essai précédent (notez bien qu'il ne s'agit que de simples exemples d'école auxquels il ne faut pas chercher à attribuer une signification précise) :

```
essai <int, float, int> ce1 ;
essai <int, int *, double > ce2 ;
essai <char *, int, obj>  ce3 ;
```

La dernière suppose bien sûr que le type obj a été préalablement défini (il peut s'agir d'un type classe).

Il est même possible d'utiliser comme paramètre de type effectif un type instancié à l'aide d'un patron de classe ; par exemple, si nous disposons du patron de classes nommé point, tel qu'il a été défini dans le paragraphe précédent, nous pouvons déclarer :

```
essai <float, point<int>, double> ce4 ;
essai <point<int>, point<float>, char *> ce5 ;
```

Remarques

1) Les problèmes de correspondance exacte rencontrés dans le cas des patrons de fonctions n'existent plus pour les patrons de classes (du moins pour les paramètres de types étudiés ici). En effet, dans le cas des patrons de fonctions, l'instanciation se basait non pas sur la liste des paramètres indiqués à la suite du mot clé template, mais sur la liste des paramètres de l'en-tête de la fonction ; dans ce cas, un même nom (muet) pouvait apparaître deux fois et il y avait donc risque d'absence de correspondance.

2) Il est tout à fait possible qu'un argument formel (figurant dans l'en-tête) d'une fonction patron soit une classe patron. En voici un exemple, dans lequel nous supposons défini le patron de classes nommé point (ce peut être le précédent) :

```
template <class T> void fct (point<T>)
{ .....
}
```

Lorsqu'il devra instancier une fonction fct pour un type T donné, le compilateur instanciera également (si cela n'a pas encore été fait) la classe patron point< T> .

3) Comme dans le cas des patrons de fonctions, on peut rencontrer des difficultés lorsque l'on doit initialiser (au sein de fonctions membre) des variables dont le type figure en paramètre puisque alors, il peut, suivant les cas, s'agir d'un type de base ou, au contraire, d'un type classe. Là encore, la nouvelle syntaxe d'initialisation des types standard (présentée dans le paragraphe 2.4 du précédent chapitre) permet de résoudre le problème.

4) Un patron de classes peut comporter des membres (donnée ou fonction) statiques. Dans ce cas, il faut savoir que chaque instance de la classe dispose de son propre jeu de membres statiques : on est en quelque sorte "statique au niveau de l'instance et non au niveau du patron". Ceci est logique dans la mesure où le patron de classes n'est qu'un moule utilisé pour instancier différentes classes ; plus précisément, un patron de classes peut toujours être remplacé par autant de définitions différentes de classes que de classes instanciées.

3. LES PARAMÈTRES EXPRESSION D'UN PATRON DE CLASSES

Un patron de classes peut comporter des paramètres expression. Bien qu'il s'agisse, ici encore, d'une notion voisine de celle présentée pour les patrons de fonctions, certaines différences importantes existent ; en particulier les valeurs effectives d'un paramètre expression devront obligatoirement être constantes dans le cas des classes.

3.1 Exemple

Supposez que nous souhaitions définir une classe tableau susceptible de manipuler des tableaux d'objets d'un type quelconque. Il vient tout naturellement à l'esprit l'idée d'en faire une classe patron possédant un paramètre de type. Il est également possible de prévoir un second paramètre permettant de préciser le nombre d'éléments du tableau.

Dans ce cas, la création de notre classe se présentera ainsi :

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   // .....
} ;
```

La liste de paramètres (template < ... >) comporte deux paramètres de nature totalement différente :

- un paramètre (désormais classique) de type, introduit par le mot clé class,
- un "paramètre expression" de type int ; on précisera sa valeur lors de la déclaration d'une instance particulière de la classe tableau.

Par exemple, avec la déclaration :

```
tableau <int, 4> ti ;
```

nous déclarerons une classe nommée ti correspondant finalement à la déclaration suivante :

```
class ti
{  int tab [4] ;
  public :
   // .....
} ;
```

Voici un exemple complet de programme définissant un peu plus complètement une telle classe patron nommée tableau ; nous l'avons simplement dotée de l'opérateur [] et d'un constructeur (sans arguments) qui ne se justifie que par le fait qu'il affiche un message approprié. Nous avons instancié des "tableaux" d'objets de type point (ici, point est à nouveau une classe "ordinaire" et non une classe patron).

```
#include <iostream.h>
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   tableau () { cout << "construction tableau \n" ; }
   T & operator [] (int i)
    { return tab[i] ;
    }
} ;
class point
{  int x, y ;
  public :
   point (int abs=1, int ord=1 )   // ici init par défaut à 1
     { x=abs ; y=ord ;
       cout << "constr point " << x << " " << y << "\n" ;
     }
   void affiche () { cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;
main()
```

```
{  tableau <int,4> ti ;
   int i ; for (i=0 ; i<4 ; i++) ti[i] = i ;
   cout << "ti : " ;
   for (i=0 ; i<4 ; i++) cout << ti[i] << " " ;
   cout << "\n" ;
   tableau <point, 3> tp ;
   for (i=0 ; i<3 ; i++) tp[i].affiche() ;
}
```

```
construction tableau
ti : 0 1 2 3
const point 1 1
const point 1 1
const point 1 1
construction tableau
coordonnées : 1 1
coordonnées : 1 1
coordonnées : 1 1
```

Exemple de classe patron comportant un paramètre expression

Remarque :

Notre classe tableau, telle qu'elle est présentée ici n'a pas véritablement d'intérêt pratique. En effet, on obtiendrait le même résultat en déclarant de simples tableaux d'objets, par exemple int ti[4] au lieu de tableau <int,4> ti. En fait, il ne s'agit ici que d'un cadre initial qu'on peut compléter à loisir. Par exemple, on pourrait facilement y ajouter un contrôle d'indice en adaptant la définition de l'opérateur [] ; on pourrait également prévoir d'initialiser les éléments du tableau. C'est d'ailleurs ce que nous aurons l'occasion de faire dans le paragraphe 7 où nous utiliserons notre patron tableau pour manipuler des tableaux à plusieurs indices.

3.2 D'une manière générale

On peut faire apparaître autant de paramètres expression qu'on le désire dans une liste de paramètres d'un patron de classes. Ces paramètres peuvent intervenir n'importe où dans la définition du patron, au même titre que n'importe quelle expression constante peut apparaître dans la définition d'une classe.

Lors de l'instanciation d'une classe comportant des paramètres expression, les paramètres effectifs correspondants doivent obligatoirement être des expressions constantes[2] d'un type rigoureusement identique (aux conversions triviales près) à celui prévu dans la liste d'arguments ; autrement dit, aucune conversion n'est possible.

Contrairement à ce qui passait pour les patrons de fonctions, il n'est pas possible de "surdéfinir" un patron de classes, c'est-à-dire de créer plusieurs patrons de même nom mais comportant une liste de paramètres (de type ou expression) différents. En conséquence, les problèmes d'ambiguïté évoqués lors de l'instanciation d'une fonction patron ne peuvent plus se poser dans le cas de l'instanciation d'une classe patron.

Sur un plan méthodologique, on pourra souvent hésiter entre l'emploi de paramètres expression et la transmission d'arguments au constructeur. Ainsi, dans notre exemple de classe tableau, nous aurions pu ne pas prévoir le paramètre expression n et, en revanche, transmettre au constructeur le nombre d'éléments

2 - Cette contrainte n'existait pas pour les paramètres expresssion des patrons de fonctions ; mais leur rôle n'était pas le même.

souhaités. Dans ce cas, une différence importante serait apparue au niveau de la gestion des emplacements mémoire correspondants aux différents éléments du tableau :

- attribution d'emplacement à la compilation (statique ou automatique suivant la classe d'allocation de l'objet de type tableau<...,...> correspondant) dans le premier cas,
- allocation dynamique par le constructeur dans le second cas.

4. SPÉCIALISATION D'UN PATRON DE CLASSES

Nous avons vu qu'il était possible de "spécialiser" certaines fonctions d'un patron de fonctions. La même possibilité existe pour les patrons de classes ; elle prend toutefois un aspect légèrement différent, à la fois au niveau de sa syntaxe et de ses possibilités comme nous le verrons après un exemple d'introduction.

4.1 Exemple de spécialisation d'une fonction membre

Un patron de classes définit une famille de classes dans laquelle chaque classe comporte à la fois sa définition et la définition de ses fonctions membre. Ainsi, toutes les fonctions membre de nom donné réalisent le même algorithme. Si l'on souhaite adapter une fonction membre à une situation particulière, il est possible d'en fournir une nouvelle.

Voici un exemple qui reprend le patron de classes point défini dans le premier paragraphe. Nous y avons spécialisé la fonction affiche dans le cas du type char, afin qu'elle affiche non plus des caractères mais des nombres entiers.

```
include <iostream.h>
  // création d'un patron de classe
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)
   {  x = abs ; y = ord ;
   }
   void affiche () ;
} ;
  // définition de la fonction affiche
template <class T> void point<T>::affiche ()
{
  cout << "Coordonnées : " << x << " " << y << "\n" ;
}
  // ajout d'une fonction affiche spécialisée pour les caractères
void point<char>::affiche ()
{
  cout << "Coordonnées : " << (int)x << " " << (int)y << "\n" ;
}
main ()
{
   point <int> ai (3, 5) ;         ai.affiche () ;
   point <char> ac ('d', 'y') ;    ac.affiche () ;
   point <double> ad (3.5, 2.3) ; ad.affiche () ;
```

```
}
```

```
coordonnées : 3 5
coordonnées : 100 121
coordonnées : 3.5 2.3
```

Exemple de spécialisation d'une fonction membre d'une classe patron

Notez qu'il nous a suffi d'écrire l'en-tête de affiche sous la forme :

```
void point<char>::affiche ()
```

pour préciser au compilateur qu'il devait utiliser cette fonction à la place de la fonction affiche du patron point, c'est-à-dire à la place de l'instance point< char> .

4.2 D'une manière générale

a) On peut spécialiser pour les valeurs de tous les paramètres

Dans notre exemple, la classe patron point ne comportait qu'un paramètre de type. Il est possible de spécialiser une fonction membre en se basant sur plusieurs paramètres de type, ainsi que sur des valeurs de paramètres expression (bien que cette dernière possiblité nous paraisse d'un intérêt limité). Par exemple, considérons le patron tableau défini dans le paragraphe 3.1 :

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   tableau () { cout << "construction tableau \n" ; }
   //  .....
} ;
```

Nous pouvons écrire une version spécialisée de son constructeur pour les tableaux de 10 éléments de type point (il ne s'agit vraiment que d'un exemple d'école !) en procédant ainsi :

```
tableau<point,10>::tableau (...) { ... }
```

b) On peut spécialiser une fonction membre ou une classe

Dans nos précédents exemples, nous avons spécialisé une fonction membre d'un patron. En fait, on peut indifféremment :

- spécialiser une ou plusieurs fonctions membre, sans modifier la définition de la classe elle-même (ce sera la situation la plus fréquente),
- spécialiser la classe elle-même, en en fournissant une nouvelle définition ; cette deuxième possibilité pouvant s'accompagner de la spécialisation de certaines fonctions membres.

Par exemple, après avoir défini le patron template < class T> class point (comme dans le paragraphe 4.1), nous pourrions définir une version spécialisée de la classe point pour le type char, c'est-à-dire une version appropriée de l'instance point< char> , en procédant ainsi :

```
class point <char>
{ // nouvelle définition
}
```

Nous pourrions également indifféremment définir des versions spécialisées de certaines des fonctions membre de point< char> en procédant comme précédemment ou ne pas en définir, auquel cas l'on ferait appel, dans ce cas, aux fonctions membre du patron.

c) Spécialisation partielle de patrons de classes

Nous avons déjà été amené à parler de spécialisation partielle dans le cas de patrons de fonctions. La même possibilité existe théoriquement pour les patrons de classe dans la norme ANSI (mais pas dans la version 3), bien qu'elle s'exprime de façon différente. En voici un exemple :

```
template <class T, class U> class A        { ..... } ;    // patron 1
template <class T>          class A<T, T*> { ..... } ;    // patron 2
```

Une déclaration telle que A<int, float> a1 utilisera le patron 1, tandis que A<int, int*> a2 utilisera le patron 2 moins spécialisé.

En pratique, la remarque effectuée à propos des spécialisations partielles de patrons de fonctions s'applique encore ici, à savoir que toutes les implémentations ne traitent pas encore convenablement ces situations. Nous n'entrerons pas plus dans les détails.

5. PARAMETRES PAR DEFAUT

Dans la définition d'un patron de classes, il est possible (depuis la norme ANSI seulement) de spécifier des valeurs par défaut pour certains paramètres, suivant un mécanisme semblable à celui qui est utilisé pour les paramètres de fonctions usuelles. Voici quelques exemples :

```
template <class T, class U=float> class A { ..... } ;
template <class T, int=3>         class B { ..... } ;
   .....
A<int,long> a1 ;      /* instanciation usuelle           */
A<int> a2 ;           /* équivaut à A<int, float> a2 ;   */
B<int, 3> b1 ;        /* instanciation usuelle           */
B<int> b2 ;           /* équivaut à B<int, 3> b2 ;       */
```

Remarque :

La notion de paramètres par défaut n'a pas de signification pour les patrons de fonctions.

6. PATRONS DE FONCTIONS MEMBRE

Le mécanisme de définition de patrons de fonctions peut s'appliquer à une fonction membre d'une classe ordinaire, comme dans cet exemple :

```
class A
{ .....
  template <class T> void fct (T a) { ..... }
  .....
} ;
```

En théorie, la norme (mais pas la version 3) permet de l'appliquer à une fonction membre d'une classe patron, comme dans cet exemple :

```
template <class T> class A
{ .....
  template <class U> void fct (U x, T y)     /* ici le type T est utilisé, mais
*/
     { ..... }                               /* il pourrait ne pas l'être
*/
  .....
} ;
```

Dans ce dernier cas, l'instanciation de la bonne fonction fct se basera, à la fois sur la classe à laquelle elle appartient, ainsi que sur la nature de son premier argument.

Cette fois encore, il faut remarquer que toutes les implémentations ne traitent pas encore convenablement ces deux situations et nous n'entrerons pas plus dans les détails.

7. IDENTITÉ DE CLASSES PATRON

Nous avons déjà vu que l'opérateur d'affectation peut s'appliquer à deux objets d'un même type. Le terme "même type" est parfaitement défini, tant que l'on n'utilise pas d'instances de patron de classes : deux objets sont de même type s'ils sont déclarés avec le même nom de classe. Mais que devient cette définition dans le cas d'objets dont le type est une instance particulière d'un patron de classe ?

En fait, deux classes patron correspondront à un même type si leurs paramètres de type correspondent exactement au même type et si leurs paramètres expression ont la même valeur.

Ainsi (en supposant que nous disposions du patron tableau défini dans le paragraphe 3.1), avec ces déclarations :

```
tableau <int, 12> t1 ;
tableau <float, 12> t2 ;
```

vous n'aurez pas le droit d'écrire :

```
t2 = t1 ;     // incorrect car valeurs différentes du premier paramètre (float
et int)
```

De même, avec ces déclarations :

```
tableau <int, 15> ta ;
tableau <int, 20> tb ;
```

vous n'aurez pas le droit d'écrire :

```
ta = tb ;     // incorrect car valeurs différentes du second paramètre (15 et
20)
```

Ces règles, apparemment restrictives, ne servent en fait qu'à assurer un bon fonctionnement de l'affectation, qu'il s'agisse de l'affectation par défaut (membre à membre : il faut donc bien disposer exactement des mêmes membres dans les deux objets) ou de l'affectation surdéfinie (pour que cela fonctionnne toujours, il faudrait compter sur le concepteur du patron de classe pour qu'il prévoie toutes les combinaisons possibles et, de plus, être sûr qu'une éventuelle spécialisation ne risque pas de pertuber les choses...).

Certes, dans le premier cas (t2= t1), une conversion int-> float nous aurait peut-être convenu mais pour que le compilateur puisse la mettre en œuvre, il faudrait qu'il "sache" qu'une classe tableau< int, 10> ne comporte que des membres de type int, qu'une classe tableau< float, 10> ne comporte que des membres de type float, que les deux classes ont le même nombre de membres donnée..

8. CLASSES PATRON ET DÉCLARATIONS D'AMITIÉS

L'existence des patrons de classes introduit de nouvelles possibilités de déclaration d'amitié.

Au sein d'un patron de classe, on peut effectuer trois sortes de déclarations d'amitié.

8.1 Déclaration de classes ou fonctions "ordinaires" amies

La démarche reste celle que nous avons déjà rencontrée dans le cas des classes ordinaires Par exemple, si A est une classe ordinaire et fct une fonction ordinaire :

```
template <class T>
class essai
{  int x ;
  public :
    friend class A ;          // A est amie de toute instance du patron essai
    friend int fct (float) ;  // fct est amie de toute instance du patron essai
    ...
} ;
```

8.2 Déclaration d'instances particulières de classes patron ou de fonctions patron

En fait, cette possibilité peut prendre deux aspects différents suivant que les paramètres utilisés pour définir l'instance concernée sont des paramètres effectifs ou muets (définis dans la liste de paramètres du patron de classe).

Si nous supposons que point est une classe patron définie ainsi :

```
template <class T> class point { ... } ;
```

et fct une fonction patron définie ainsi :

```
template <class T> int fct (T) { ... }
```

Voici un exemple illustrant le premier aspect :

```
template <class T, class U>
class essai1
{  int x ;
  public :
   friend class point<int> ;      // la classe patron point<int> est amie
                                  // de toutes les instances de essai1
   friend int fct (double) ;      // la fonction patron int fct (double
                                  // de toutes les instances de essai1
   ...
} ;
```

Voici un exemple illustrant le second aspect :

```
template <class T, class U>
class essai2
{  int x ;
  public :
   friend class point<T> ;
   friend int fct (U) ;
}
```

Notez bien, que dans le second cas, on établit un "couplage" entre la classe patron générée par le patron essai2 et les déclarations d'amitiés correspondantes. Par exemple, pour l'intance essai2 < int, double> , les déclarations d'amitié porteront sur point< int> et int fct (double).

8.3 Déclaration d'un autre patron de fonctions ou de classes

Voici un exemple faisant appel aux mêmes patrons point et fct que ci-dessus :

```
template <class T, class U>
class essai2
{  int x;
  public :
   template <class X> friend class point <X> ;
   template <class X> friend int fct (point <X>) ;
}
```

Cette fois, toutes les instances du patron point sont amies de n'importe quelle instance du patron essai2. De même toutes les instances du patron de fonctions fct sont amies du n'importe instance du patron essai2.

9. UN EXEMPLE DE CLASSE TABLEAU À DEUX INDICES

Nous avons vu à plusieurs reprises comment surdéfinir l'opérateur [] au sein d'une classe tableau. Néanmoins, nous nous sommes toujours limité à des tableaux à un indice.

Ici, nous allons voir comment, une fois défini un patron de tableau comme nous l'avons fait précédemment (donc à un indice), il est très facile, par le simple jeu de la composition des patrons, de l'appliquer à un tableau à deux indices (ou davantage).

Si nous considérons pour l'instant la classe tableau définie de cette façon simplifiée :

```
template <class T, int n> class tableau
{  T tab [n] ;
  public :
   T & operator [] (int i)       // opérateur []
     { return tab[i] ;
     }
} ;
```

nous pouvons tout à fait déclarer :

```
    tableau <tableau<int,2>,3> t2d ;
```

Dans ce cas, t2d est, en effet, un tableau de 3 éléments ayant chacun le type tableau < int,2> ; autrement dit, chacun de ces 3 éléments est lui-même un tableau de 2 entiers.

Une notation telle que t2d [1] [2] a un sens ; elle représente la référence au troisième élément de t2d [1], c'est-à-dire au troisième élément du deuxième tableau de deux entiers de t2d.

Voici un exemple complet (mais toujours simplifié) illustrant ceci ; nous y avons simplement ajouté artificiellement un constructeur afin d'obtenir une trace des différentes constructions.

```
  // implémentation d'un tableau à deux dimensions
#include <iostream.h>
template <class T, int n> class tableau
{
   T tab [n] ;
  public :
   tableau ()                    // constructeur
     {cout << "construction tableau à " << n << " éléments\n" ;
     }
   T & operator [] (int i)       // opérateur []
     { return tab[i] ;
     }
} ;

main()
{
    tableau <tableau<int,2>,3> t2d ;
    t2d [1] [2] = 15 ;
    cout << "t2d [1] [2] = " << t2d [1] [2] << "\n" ;
```

```
    cout << "t2d [0] [1] = " << t2d [0] [1] << "\n" ;
}
```

```
construction tableau à 2 éléments
construction tableau à 2 éléments
construction tableau à 2 éléments
construction tableau à 3 éléments
t2d [1] [2] = 15
t2d [0] [1] = 2226
```

Utilisation du patron tableau pour manipuler des tableaux à deux indices (1)

On notera bien que notre patron tableau est, a priori, un tableau à un indice ; c'est simplement la manière dont on l'utilise qui permet de l'appliquer à des tableaux à un nombre quelconque d'indices.

Manifestement, cet exemple est trop simpliste ; d'ailleurs, tel quel, il n'apporte rien de plus qu'un banal tableau. Pour le rendre plus réaliste, nous allons prévoir :

- de gérer les "débordements d'indices" : ici, nous nous contenterons d'afficher un message et de "faire comme si" l'utilisateur avait fourni un indice nul,
- d'initialiser, lors de sa construction, tous les éléments du tableau : nous utiliserons pour ce faire la valeur 0 ; encore faut-il que la chose soit possible, c'est-à-dire que, quel que soit le type T des éléments du tableau, on puisse leur affecter la valeur 0 ; cela signifie qu'il doit exister une conversion de T en int ; il est facile de la réaliser avec un constructeur à un élément de type int ; du même coup, cela permettra de prévoir une valeur initiale lors de la déclaration d'un tableau (par sécurité, nous prévoirons la valeur 0 par défaut).

Voici notre nouvelle classe ainsi modifiée et un exemple d'utilisation :

```
  // implémentation d'un tableau 2d avec test débordement d'indices
#include <iostream.h>
template <class T, int n> class tableau
{  T tab [n] ;
   int limite ;          // nombre d'éléments du tableau
  public :


   tableau (int init=0)
     {  int i ;
        for (i=0 ; i<n ; i++) tab[i] = init ;
                     // il doit exister un constructeur à un argument
                     // pour le cas où tab[i] est un objet
        limite = n-1 ;
```

```
          cout << "appel constructeur tableau de taille " << n
               << " init = " << init << "\n" ;
      }
    T & operator [] (int i)
    { if (i<0 || i>limite) { cout << "--débordement " << i << "\n" ;
                             i=0 ;  // choix arbitraire
                           }
      return tab[i] ;
    }
} ;
main()
{   tableau <tableau<int,3>,2> ti ;          // pas d'initialisation
    tableau <tableau<float,4>,2> td (10) ;  // initialisation à 10
    ti [1] [6] = 15 ;
    ti [8] [-1] = 20 ;
    cout << ti [1] [2] << "\n" ;   // élément initialisé à valeur par défaut
(0)
    cout << td [1] [0] << "\n" ;   // élément initialisé explicitement
}
```

```
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 3 init = 0
appel constructeur tableau de taille 2 init = 0
appel constructeur tableau de taille 4 init = 0
appel constructeur tableau de taille 4 init = 0
appel constructeur tableau de taille 4 init = 10
appel constructeur tableau de taille 4 init = 10
appel constructeur tableau de taille 2 init = 10
--débordement 6
--débordement 0
--débordement -1
0
10
```

Utilisation du patron tableau pour manipuler des tableaux à deux indices (2)

Remarque :

Si vous examinez bien les messages de construction des différents tableaux, vous constaterez que l'on en obtient deux fois plus que prévu pour les tableaux à un indice. L'explication réside dans l'instruction tab[i] = init du constructeur tableau. En effet, lorsque tab[i] désigne un élément de type de base, il y a simplement conversion de la valeur entière init dans ce type de base. En revanche, lorsque l'on a affaire à un objet de type T (ici T est de la forme tableau<...>), cette instruction provoque l'appel du constructeur tableau(int) pour créer un objet temporaire de ce type. Ceci se voit très clairement dans le cas du tableau td pour lequel on trouve une construction d'un tableau temporaire initialisé avec la valeur 0 et une construction d'un tableau initialisé avec la valeur 10.

XII. LA TECHNIQUE DE L'HÉRITAGE

Comme nous l'avons déjà évoqué dans le premier chapitre, le concept d'héritage (on parle également de classes dérivées) constitue l'un des fondements de la P.O.O. En particulier, il est à la base des possibilités de réutilisation de composants logiciels (en l'occurrence, de classes). En effet, il vous autorise à définir une nouvelle classe, dite "dérivée", à partir d'une classe existante dite "de base". La classe dérivée "héritera" des "potentialités" de la classe de base, tout en lui en ajoutant de nouvelles, et cela sans qu'il soit nécessaire de remettre en question la classe de base. Il ne sera pas utile de la recompiler ; d'ailleurs, il ne sera même pas nécessaire de disposer du programme source correspondant (exception faite de sa déclaration).

Cette technique va donc permettre de développer de nouveaux outils en se fondant sur un certain acquis, ce qui justifie le terme d'héritage. Bien entendu, plusieurs classes pourront être dérivées d'une même classe de base ; d'autre part, l'héritage n'est pas limité à un seul niveau : une classe dérivée pouvant devenir à son tour classe de base pour une autre classe. On voit ainsi apparaître la notion d'héritage comme outil de spécialisation croissante.

Qui plus est, nous verrons que C++ (depuis la version 2.0) autorise l'héritage multiple dans lequel une classe peut être dérivée de plusieurs classes de base.

Nous commencerons, dans ce chapitre, par vous présenter la mise en œuvre de l'héritage en C++ sur un exemple très simple. Nous examinerons ensuite comment, à l'image de ce qui se passait dans le cas d'objets membre, C++ offre un mécanisme intéressant de transmission d'information entre constructeurs (de la classe dérivée et de la classe de base). Puis nous verrons la souplesse que présente le C++ en matière de contrôle des accès de la classe dérivée aux membres de la classe de base (aussi bien au niveau de la conception de la classe de base que celle de la classe dérivée).

Nous aborderons ensuite les problèmes de compatibilité entre une classe de base et une classe dérivée, tant au niveau des objets eux-mêmes que des pointeurs sur ces objets ou des références à ces objets. Nous examinerons alors ce qu'il advient du constructeur de recopie, de l'opérateur d'affectation et des patrons de classes, dans le cadre de l'héritage.

Enfin, après avoir fait la distinction entre héritage simple et héritage multiple, nous apprendrons à exploiter concrètement une classe dérivée.

Quant à l'héritage multiple, il fera l'objet du chapitre suivant.

1. MISE EN ŒUVRE DE L'HÉRITAGE EN C++

Tout d'abord, nous allons vous exposer les bases de la mise en œuvre de l'héritage en C++ sur un exemple simple ne faisant pas intervenir de constructeur ou de destructeur et où le contrôle des accès est limité.

Considérons la première classe point que nous avions définie dans le chapitre V, dont nous fournissons à nouveau la définition ci-dessous :

```
#include <iostream.h>
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
              /* déclaration des membres publics */
 public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
     /* ----- Définition des fonctions membres de la classe point ---- */
void point::initialise (int abs, int ord)
{   x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{   x += dx ; y += dy ;
}
void point::affiche ()
{   cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Une classe de base : point

Supposez que nous ayons besoin de définir un nouveau type classe nommé pointcol, destiné à manipuler des points colorés d'un plan. Un tel point coloré peut être défini par ses coordonnées (comme un objet de type point), auxquelles on adjoint une information de couleur (par exemple de type char). Dans ces conditions, nous pouvons être tenté de définir pointcol comme une classe dérivée de point. Si nous prévoyons (pour l'instant) une fonction membre spécifique à pointcol, nommée colore, destinée à attribuer une couleur à un point coloré, voici ce que pourrait être la déclaration de pointcol (la fonction colore est ici "en ligne").

```
#include <point.h>
class pointcol : public point        // pointcol dérive de point
{  short couleur ;
 public :
```

```
    void colore (short cl)
        { couleur = cl ; }
} ;
```

Une classe pointcol, dérivée de point

Notez la déclaration :

```
class pointcol : public point
```

Elle spécifie que pointcol est une classe dérivée de la classe (de base) point. De plus, le mot public signifie que **les membres publics de la classe de base (point) seront des membres publics de la classe dérivée (pointcol)** ; cela correspond à l'idée la plus fréquente que l'on peut avoir de l'héritage, sur le plan général de la P.O.O. Nous verrons plus loin, dans le paragraphe consacré au contrôle des accès, à quoi conduirait l'omission du mot public.

Notre classe pointcol ainsi définie, nous pouvons déclarer des objets de type pointcol de manière usuelle :

```
pointcol p, q ;
```

Chaque objet de type pointcol peut alors faire appel :

- aux méthodes publiques de pointcol (ici colore),
- aux méthodes publiques de la classe de base point (ici init, deplace et affiche).

Voici un programme illustrant ces possibilités. Vous n'y trouverez pas la liste de la classe point car nous nous sommes placé dans les conditions habituelles d'utilisation d'une classe déjà au point ; plus précisément, nous supposons que nous disposons :

- d'un module objet relatif à la classe point : il est nécessaire de l'incorporer au moment de l'édition de liens,
- d'un fichier nommé ici point.h[1], contenant la déclaration de la classe point.

```
#include <iostream.h>
#include <point.h>
class pointcol : public point        // pointcol dérive de point
{  short couleur ;
 public :
   void colore (short cl)  { couleur = cl ; }
} ;
main()
{ pointcol p ;
  p.initialise (10,20) ; p.colore (5) ;
  p.affiche () ;
  p.deplace (2,4) ;
  p.affiche () ;
```

1 - Son extension pourra varier suivant l'environnement utilisé. De plus, suivant sa localisation, on utilisera l'une ou l'autre des deux syntaxes de la directive #include (<.....> ou ".....").

```
}

Je suis en 10 20
Je suis en 12 24
```

Exemple d'utilisation d'une classe pointcol, dérivée de point

2. UTILISATION, DANS UNE CLASSE DERIVÉE, DES MEMBRES DE LA CLASSE DE BASE

Comme nous l'avons dit, l'exemple précédent n'était destiné qu'à montrer comment s'exprime l'héritage en C++, sans chercher à en explorer toutes les possibilités, notamment en matière de contrôle des accès. Dans ce domaine, nous avons simplement appris que, grâce à l'emploi du mot public, les membres publics de point étaient également membres publics de pointcol ; c'est ce qui nous a permis d'y faire appel, au sein de la fonction main (par exemple dans l'instruction p.initialise(10, 20)).

Or, la classe pointcol, telle que nous l'avons définie, souffre de lacunes. Par exemple, lorsque nous appelons affiche pour un objet de type pointcol, nous n'obtenons aucune information sur sa couleur. Une première façon d'améliorer cette situation consiste à écrire une nouvelle fonction membre publique de pointcol, censée afficher à la fois les coordonnées et la couleur. Appelons-la pour l'instant affichec (nous verrons plus tard qu'il est possible de l'appeler également affiche).

A ce niveau, vous pourriez penser définir affichec de la manière suivante :

```
void affichec ()
{ cout << "Je suis en " << x << " " << y << "\n" ;
  cout << "     et ma couleur est : " << couleur << "\n" ;
}
```

Mais alors cela signifierait que la fonction affichec, membre de pointcol, aurait accès aux membres privés de point. Ceci serait contraire au principe d'encapsulation : en effet, il deviendrait alors possible d'écrire une fonction accédant directement[2] aux données privées d'une classe, simplement en créant une classe dérivée ! D'où la règle adoptée par C++ :

Une classe dérivée n'a pas accès aux membres privés de sa classe de base[3].

En revanche, rien n'empêche à une classe dérivée d'accéder à n'importe quel membre public de sa classe de base. Ainsi, dans le cas qui nous préoccupe, si notre fonction membre affichec ne peut pas accéder directement aux données privées x et y de la classe point, elle peut néanmoins faire appel à la fonction affiche de cette même classe. D'où une définition possible de affichec.

```
void pointcol::affichec ()
{  affiche () ;
   cout << "     et ma couleur est : " << couleur << "\n" ;
}
```

2 - C'est-à-dire sans passer par l'interface obligatoire constituée par les fonctions membre publiques.

3 - Mais nous verrons que l'existence de membres protégés permet de modifier cela.

Une fonction d'affichage pour un objet de type pointcol

Notez bien comment, au sein de affichec, nous avons fait directement appel à affiche, **sans avoir à spécifier à quel objet cette fonction devait être appliquée** : par convention, il s'agit de celui ayant appelé affichec. Nous retrouvons la même règle que celle qui s'applique dans le cas de fonctions membre d'une même classe. En fait, il faut désormais considérer ici que affiche est une fonction membre de pointcol[4].

D'une manière analogue, nous pouvons définir dans pointcol, une nouvelle fonction d'initialisation nommée initialisec, chargée d'attribuer des valeurs aux données x, y et couleur, à partir de trois valeurs reçues en argument.

```
void pointcol::initialisec (int abs, int ord, short cl)
{  initialise (abs, ord) ;
   couleur = cl ;
}
```

Une fonction d'initialisation pour un objet de type pointcol

Voici un exemple complet de programme reprenant la définition de la classe pointcol (ici encore, nous avons supposé que la classe point était fournie séparément).

```
#include <iostream.h>
#include <point.h>
class pointcol : public point
{  short couleur ;
 public :
   void colore (short cl)
       { couleur = cl ; }
   void affichec () ;
   void initialisec (int, int, short) ;
} ;
void pointcol::affichec ()
{  affiche () ;
   cout << "     et ma couleur est : " << couleur << "\n" ;
}
void pointcol::initialisec (int abs, int ord, short cl)
{  initialise (abs, ord) ;
   couleur = cl ;
}
main()
```

4 - Ce ne serait toutefois pas le cas si affiche n'était pas une fonction publique de point.

```
{ pointcol p ;
  p.initialisec (10,20, 5) ; p.affichec () ; p.affiche () ;
  p.deplace (2,4) ;          p.affichec () ;
  p.colore (2) ;             p.affichec () ;
}
```

```
Je suis en 10 20
     et ma couleur est : 5
Je suis en 10 20
Je suis en 12 24
     et ma couleur est : 5
Je suis en 12 24
     et ma couleur est : 2
```

Une nouvelle classe pointcol et son utilisation

3. REDÉFINITION DES FONCTIONS MEMBRE

Dans notre dernier exemple de classe pointcol, nous disposions à la fois :

- dans point, d'une fonction membre nommée affiche,
- dans pointcol, d'une fonction membre nommée affichec.

Or, ces deux méthodes font un travail analogue, à savoir afficher les valeurs des données de leur classe ; dans ces conditions, on pourrait souhaiter leur donner le même nom. Ceci est effectivement possible en C++, moyennant simplement une petite précaution. En effet, il n'est alors plus possible, au sein de la fonction affiche de pointcol, d'appeler la fonction affiche de point, comme auparavant : cela provoquerait un appel récursif de la fonction affiche de pointcol. Il faut alors faire appel à l'opérateur de résolution de portée (::) pour localiser convenablement la méthode voulue (ici, on appellera point::affiche).

De manière comparable, si, pour un objet p de type pointcol, on appelle la fonction p.affiche, il s'agira de la fonction redéfinie dans pointcol. Si l'on tient absolument à utiliser la fonction affiche de la classe point, on appellera p.point::affiche.

A titre d'exemple, voici comment nous pouvons transformer l'exemple du paragraphe précédent en nommant affiche et initialise les nouvelles fonctions membre de pointcol.

```
#include <iostream.h>
#include <point.h>
class pointcol : public point
{  short couleur ;
 public :
   void colore (short cl)
       { couleur = cl ; }
   void affiche () ;                    // redéfinition de affiche de point
   void initialise (int, int, short) ;  // redéfinition de initialise de point
} ;
void pointcol::affiche ()
```

```
{  point::affiche () ;                // appel de affiche de la classe point
   cout << "     et ma couleur est : " << couleur << "\n" ;
}
void pointcol::initialise (int abs, int ord, short cl)
{  point::initialise (abs, ord) ;     // appel de initialise de la classe point
   couleur = cl ;
}

main()
{ pointcol p ;
  p.initialise (10,20, 5) ; p.affiche () ;
  p.point::affiche () ;               // pour forcer l'appel de affiche de point
  p.deplace (2,4) ;          p.affiche () ;
  p.colore (2) ;             p.affiche () ;
}
```

```
Je suis en 10 20
     et ma couleur est : 5
Je suis en 10 20
Je suis en 12 24
     et ma couleur est : 5
Je suis en 12 24
     et ma couleur est : 2
```

Une classe pointcol dans laquelle les méthodes initialise et affiche sont redéfinies

Remarque :

Bien que cela soit d'un emploi moins courant, ce que nous avons dit à propos de la redéfinition de fonctions membre s'applique tout aussi bien aux membres donnée. Plus précisément, si une classe A est définie ainsi :

```
class A
{     .....
      int a ;
      char b ;
      .....
} ;
```

Une classe B dérivée de A pourra, par exemple, définir un autre membre donnée nommé a :

```
class B : public A
{     float a ;
      .....
} ;
```

Dans ce cas, si l'objet b est de type B, b.a fera référence au membre a de type float de b. Il sera toujours possible d'accéder au membre donnée a de type int (hérité de A) par b.A::a[5].

Notez bien que le membre a défini dans B s'ajoute au membre a hérité de A ; il ne le remplace pas.

5 - En supposant, bien sûr, que les accès en question soient autorisés.

4. APPEL DES CONSTRUCTEURS ET DES DESTRUCTEURS

4.1 Rappels

Rappelons l'essentiel des règles concernant l'appel d'un constructeur ou du destructeur d'une classe (dans le cas où il ne s'agit pas d'une classe dérivée) :

- S'il existe au moins un constructeur, toute création d'un objet (par déclaration ou par new) entraînera l'appel d'un constructeur. Le choix éventuel du constructeur est réalisé en fonction des informations fournies ; si aucun constructeur ne convient, il y a erreur de compilation : il n'est donc pas possible, dans ce cas, de créer un objet sans appeler l'un de ses constructeurs.
- S'il n'existe aucun constructeur, il n'est pas possible de préciser des informations lors de la création d'un objet.
- S'il existe un destructeur, il sera appelé avant la destruction de l'objet.

4.2 La hiérarchisation des appels

Ces règles se généralisent au cas des classes dérivées, en tenant compte de l'aspect hiérarchique qu'elles introduisent. Pour fixer les idées, supposons que chaque classe possède un constructeur et un destructeur.

```
class A                          class B : public A
{     .....                            {     .....
  public :                           public :
      A (...)                                B (...)
      ~A ()                                  ~B ()
      .....                                  .....
} ;                              } ;
```

On conçoit que, pour créer un objet de type B, il faut tout d'abord créer un objet de type A, donc faire appel au constructeur de A, puis le compléter par ce qui est spécifique à B et faire appel au constructeur de B. Ce mécanisme est pris en charge par C++ : autrement dit, il n'y aura pas à prévoir dans le constructeur de B l'appel du constructeur de A.

La même démarche s'applique aux destructeurs : lors de la destruction d'un objet de type B, il y aura automatiquement appel du destructeur de B, puis appel de celui de A (les destructeurs sont bien appelés dans l'ordre inverse de l'appel des constructeurs).

4.3 Transmission d'informations entre constructeurs

Toutefois, vous voyez qu'un problème se pose dans le cas où le constructeur de A nécessite des arguments. En effet, les informations fournies lors de la création d'un objet de type B sont, a priori, destinés à son constructeur ! En fait, C++ a prévu la possibilité de spécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre à un constructeur de la classe de base. Le mécanisme est le même que celui que nous vous avons exposé dans le cas des objets membre (paragraphe 7 du chapitre VII). Par exemple, si l'on a ceci :

```
class point                      class pointcol : public point
{     .....                      {     .....
```

```
  public :                        public :
      point (int, int) ;                  pointcol (int, int, char) ;
      .....                               .....
} ;                             } ;
```

et que l'on souhaite que pointcol retransmette à point les deux premières informations reçues, on écrira son en-tête de cette manière :

```
pointcol (int abs, int ord, char cl) : point (abs, ord)
```

Le compilateur mettra en place la transmission au constructeur de point des informations abs et ord correspondant (ici) aux deux premiers arguments de pointcol. Ainsi, la déclaration :

```
pointcol a (10, 15, 3) ;
```

entraînera :

- l'appel de point qui recevra les arguments 10 et 15,
- l'appel de pointcol qui recevra les arguments 10, 15 et 3.

En revanche, la déclaration :

```
pointcol q (5, 2)
```

sera rejetée par le compilateur puisqu'il n'existe aucun constructeur pointcol à deux arguments.

Bien entendu, il reste toujours possible de mentionner des arguments par défaut dans pointcol, par exemple :

```
pointcol (int abs = 0, int ord = 0, char cl = 1) : point (abs, ord)
```

Dans ces conditions, la déclaration :

```
pointcol b (5) ;
```

entraînera :

- l'appel de point avec les arguments 5 et 0,
- l'appel de pointcol avec les arguments 5, 0 et 1.

Notez que la présence éventuelle d'arguments par défaut dans point n'a aucune incidence ici (mais on peut les avoir prévus pour les objets de type point).

4.4 Exemple

Voici un exemple complet de programme illustrant cette situation : les classes point et pointcol y ont été limitées à leurs constructeurs et destructeurs (ce qui leur enlèverait, bien sûr, tout intérêt en pratique).

```
#include <iostream.h>
         // ************ classe point *********************
class point
{  int x, y ;
  public :
```

```
    point (int abs=0, int ord=0)        // constructeur de point ("inline")
      { cout << "++ constr. point :    " << abs << " " << ord << "\n" ;
        x = abs ; y =ord ;
      }

   ~point ()                            // destructeur de point ("inline")
     { cout << "-- destr.  point :    " << x << " " << y << "\n" ;
     }
} ;
         // ************ classe pointcol ******************
class pointcol : public point
{  short couleur ;
  public :
   pointcol (int, int, short) ;         // déclaration constructeur pointcol
   ~pointcol ()                         // destructeur de pointcol ("inline")
     { cout << "-- dest. pointcol - couleur : " << couleur << "\n" ;
     }
} ;
pointcol::pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
{ cout << "++ constr. pointcol : " << abs << " " << ord << " " << cl << "\n" ;
  couleur = cl ;
}

         // ************ programme d'essai ****************
main()
{  pointcol a(10,15,3) ; pointcol b (2,3) ;         // objets
   pointcol c (12) ;     pointcol d ;               // automatiques
   pointcol * adr ;
   adr = new pointcol (12,25) ;                     // objet dynamique
   delete adr ;
}
```

```
++ constr. point :    10 15
++ constr. pointcol : 10 15 3
++ constr. point :    2 3
++ constr. pointcol : 2 3 1
++ constr. point :    12 0
++ constr. pointcol : 12 0 1
++ constr. point :    0 0
++ constr. pointcol : 0 0 1
++ constr. point :    12 25
++ constr. pointcol : 12 25 1
-- dest. pointcol - couleur : 1
-- destr.  point :    12 25
-- dest. pointcol - couleur : 1
-- destr.  point :    0 0
-- dest. pointcol - couleur : 1
-- destr.  point :    12 0
-- dest. pointcol - couleur : 1
-- destr.  point :    2 3
-- dest. pointcol - couleur : 3
-- destr.  point :    10 15
```

Appel des constructeurs et destructeurs de la classe de base et de la classe dérivée

Remarque :

Dans le message affiché par ~pointcol, vous auriez peut être souhaité voir apparaître les valeurs de x et de y. Or, ceci n'est pas possible, du moins telle que la classe point a été conçue. En effet, un membre d'une classe dérivée n'a pas accès aux membres privés de la classe de base. Nous reviendrons sur cet aspect qui s'avère fondamental dans la conception de classes "réutilisables".

4.5 D'une manière générale

Nous venons d'examiner un cas qu'on pourrait qualifier d'usuel : la classe de base et la classe dérivée possédaient leur constructeur (au moins un).

Si la classe de base ne possède pas de constructeur, aucun problème particulier ne se pose. Il en va de même si elle ne possède pas de destructeur.

En revanche, si la classe dérivée ne possède pas de constructeur, alors que la classe de base en comporte, on va voir se poser à nouveau le problème de la transmission des informations attendues par le constructeur de la classe de base. Comme celles-ci ne peuvent plus provenir du constructeur de la classe dérivée, on comprend que la seule situation acceptable soit celle où la classe de base dispose d'un constructeur sans argument. Dans les autres cas, on aboutira à une erreur de compilation.

Par ailleurs, lorsque l'on mentionne les informations à transmettre à un constructeur de la classe de base, on n'est pas obligé de se limiter, comme nous l'avons fait jusqu'ici, à des noms d'arguments. On peut employer n'importe quelle expression. Par exemple, bien que cela n'ait guère de sens ici, nous pourrions écrire :

```
pointcol (int abs, int ord, char cl) : point (abs + ord, abs - ord)
```

Remarque :

Le cas du constructeur de recopie sera examiné un peu plus loin car sa bonne mise en œuvre nécessite la connaissance des possibilités de conversion implicite d'une classe dérivée en un classe de base que nous n'avons pas encore examinées.

5. CONTRÔLE DES ACCÈS

Comme nous l'avons indiqué dans le premier paragraphe, nous n'avons examiné jusqu'ici que la situation d'héritage la plus naturelle, c'est-à-dire celle dans laquelle :

- la classe dérivée[6] a accès aux membres publics de la classe de base,
- les "utilisateurs[7]" de la classe dérivée ont accès à ses membres publics, ainsi qu'aux membres publics de sa classe de base.

Comme nous allons le voir maintenant, C++ permet d'intervenir en partie sur ces deux sortes d'autorisation d'accès, et ceci à deux niveaux :

---

6 - Sous-entendu : toute fonction membre d'une classe dérivée.

7 - Sous-entendu : tout objet du type de la classe dérivée.

Lors de la conception de la classe de base : en effet, en plus des statuts publics et privés que nous connaissons, il existe un troisième statut dit "protégé" (mot clé protected). Les membre protégés se comportent comme des membres privés pour l'utilisateur de la classe dérivée mais comme des membres publics pour la classe dérivée elle-même.

Lors de la conception de la classe dérivée, on peut restreindre les possibilités d'accès aux membres de la classe de base.

5.1 Les membres protégés

Jusqu'ici, nous avons considéré qu'il n'existait que deux "statuts" possibles pour un membre de classe :

- privé : le membre n'est accessible qu'aux fonctions membre (publiques ou privées) et aux fonctions amies de la classe,
- public : le membre est accessible non seulement aux fonctions membre ou aux fonctions amies, mais également à l'utilisateur de la classe (c'est-à-dire à n'importe quel objet du type de cette classe).

Nous avons vu comment l'emploi des mots clés public et private permettait de distinguer les membres privés des membres publics.

Il existe en fait un troisième "statut" : protégé ; il est défini par le mot clé **protected** qui s'emploie comme les deux mots clés précédents. La définition d'une classe peut, par exemple, prendre alors l'allure suivante :

```
class X
{ private :
      .....     //     partie privée
  protected :
      .....     //     partie protégée
  public :
      .....     //     partie publique
} ;
```

Les membres protégés restent inaccessibles à "l'utilisateur" de la classe, pour qui ils apparaissent analogues à des membres privés. Mais ils seront accessibles aux membres d'une éventuelle classe dérivée, tout en restant (dans tous les cas) inaccessibles aux "utilisateurs" de cette classe.

5.2 Exemple

Dans le début du paragraphe 2, nous avions évoqué l'impossibilité, pour une fonction membre d'une classe pointcol dérivée de point, d'accéder aux membres privés x et y de point. Si nous définissons ainsi notre classe point :

```
class point
{ protected :
        int x, y ;
  public :
        point ( ... ) ;
        affiche () ;
         .....
} ;
```

il devient possible de définir, dans pointcol, une fonction membre affiche de la manière suivante :

```
class pointcol : public point
{
        short couleur ;
  public :
        void affiche ()
         { cout << "Je suis en " << x << " " << y << "\n" ;
           cout << "     et ma couleur est " << couleur << "\n" ;
         }
```

5.3 Intérêt du statut protégé

Les membres **privés** d'une classe sont **définitivement inaccessibles** depuis ce que nous appellerons "l'extérieur" de la classe (objets de cette classe, ou fonctions membre d'une classe dérivée, ou objets de cette classe dérivée...). Ceci peut poser des problèmes au concepteur d'une classe dérivée, notamment dans le cas où ces membres sont des données, dans la mesure où il est contraint, comme un "banal utilisateur", de passer par "l'interface" obligatoire. De plus, cette façon de faire peut nuire à l'efficacité du code généré.

L'introduction du statut protégé constitue donc un progrès manifeste : les membres protégés restent comparables à des membres privés pour l'utilisateur de la classe, mais ils sont comparables à des membres publics pour le concepteur d'une classe dérivée (tout en restant comparables à des membres privés pour l'utilisateur de cette dernière). Néanmoins, il faut reconnaître que, du même coup, on offre les moyens de violer (consciemment) le principe d'encapsulation des données. En effet, rien n'empêche un utilisateur d'une classe comportant une partie protégée, de créer une classe dérivée contenant les fonctions appropriées permettant d'accéder aux données correspondantes. Bien entendu, il s'agit d'un viol voulu et conçu délibérément par l'utilisateur ; cela n'a plus rien à voir avec des risques de modification **accidentelle** des données.

Remarques :

1) Lorsqu'une classe dérivée possède des fonctions amies, ces dernières disposent exactement des mêmes autorisations d'accès que les fonctions membre (de la classe dérivée). En particulier, les fonctions amies d'une classe dérivée auront bien accès aux membres déclarés protégés dans sa classe de base.

2) En revanche, les déclarations d'amitié ne s'héritent pas. Ainsi, si f a été déclarée amie d'une classe A et si B dérive de A, f n'est pas automatiquement amie de B (il est bien sûr possible de prévoir une déclaration appropriée d'amitié dans B).

5.4 Action sur le statut des membres d'une classe dérivée : dérivation publique ou privée

a) Rappels concernant la dérivation publique

Nos précédents exemples faisaient intervenir la forme la plus courante de dérivation dite "dérivation publique" car introduite par le mot clé public dans la déclaration de la classe dérivée, comme dans :

```
class pointcol : public point { ... } ;
```

Dans ce cas, rappelons que :

- les membres publics de la classe de base sont accessibles à "tout le monde", c'est-à-dire à la fois aux fonctions membre et aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée,
- les membres protégés de la classe de base sont accessibles aux fonctions membre et aux fonctions amies de la classe dérivée mais pas aux utilisateurs de cette classe dérivé,
- les membres privés de la classe de base sont inaccessibles à la fois aux fonctions membre ou amies de la classe dérivée et aux utilisateurs de cette classe dérivée.

De plus, tous les membres de la classe de base conservent, dans la classe dérivée, le statut qu'ils avaient dans la classe de base. Cette remarque n'intervient qu'en cas de dérivation d'une nouvelle classe de la classe dérivée.

Voici un tableau récapitulant la situation :

Statut dans la classe de base	Accès aux fonctions membre et amies de la classe dérivée	Accès à un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée, en cas de nouvelle dérivation
public	oui	oui	public
protégé	oui	non	protégé
privé	non	non	privé

La dérivation publique

Ces possibilités peuvent être restreintes en définissant ce que l'on nomme des dérivations privées ou protégées.

b) Dérivation privée

Il est possible, moyennant l'utilisation du mot clé private au lieu du mot clé public, d'interdire à un utilisateur d'une classe dérivée l'accès aux membres publics de sa classe de base. Par exemple, avec ces déclarations :

```
class point
{     .....
  public :
      point (...) ;
      void affiche () ;
      void deplace (...) ;
      ...
} ;
```

```
class pointcol : private point
{     .....
public :
        pointcol (...) ;
     void colore (...) ;
        ...
} ;
```

Si p est de type pointcol, les appels suivants seront rejetés par le compilateur[8] :

```
p.affiche ()          ou même          p.point::affiche ()
p.deplace (...)  ou même          p.point::deplace (...)
```

alors que, naturellement, celui-ci sera accepté :

```
p.colore (...)
```

On peut, à juste titre, penser que cette technique limite l'intérêt de l'héritage. Plus précisément, le concepteur de la classe dérivée peut, quant à lui, utiliser librement les membres publics de la classe de base (comme un utilisateur ordinaire) ; en revanche, il décide de fermer totalement cet accès à l'utilisateur de la classe dérivée. On peut dire que l'utilisateur connaîtra toutes les fonctionnalités de la classe en lisant sa déclaration, sans qu'il n'ait aucunement besoin de lire celle de sa classe de base (il n'en allait pas de même dans la situation usuelle : dans nos exemples des paragraphes précédents, pour connaître l'existence de la fonction membre deplace pour la classe pointcol, il fallait connaître la déclaration de point).

Cela montre que cette technique de fermeture des accès à la classe de base ne sera employée que dans des cas bien précis, par exemple :

- lorsque toutes les fonctions utiles de la classe de base sont redéfinies dans la classe dérivée et qu'il n'y a aucune raison de laisser l'utilisateur accéder aux anciennes,
- lorsque l'on souhaite adapter l'interface d'une classe, de manière à répondre à certaines exigences ; dans ce cas, la classe dérivée peut, à la limite, ne rien apporter de plus (pas de nouvelles données, pas de nouvelles fonctionnalités) : elle fait la "même chose" que la classe de base, seule son utilisation est différente !

Remarques :

1) Dans le cas d'une dérivation privée, les membres protégés de la classe de base restent accessibles aux fonctions membre et aux fonctions amies de la classe dérivée ; en revanche, ils seront considérés comme privés pour une dérivation future.

2) Les termes de *dérivation publique* ou de *dérivation privée* seront, en toute rigueur, ambigus dans le cas d'héritage multiple ; plus précisément, il faudra alors dire, pour chaque classe de base, quel est le type de dérivation (publique ou privée).

5.5 Les possibilités de dérivation protégée (version 3)

La version 3 de C++ a introduit une possibilité supplémentaire de dérivation, dite dérivation protégée, intermédiaire entre la dérivation publique et la dérivation privée. Dans ce cas, les membres publics de la classe de base seront considérés comme protégés lors de dérivation ultérieures.

Remarque :

Ne confondez pas le mode de dérivation d'une classe par rapport à sa classe de base (publique, protégée ou privée), définie par l'un des mots public, protected ou private avec le statut des membres d'une classe (public, protégé ou privé) défini également par l'un de ces trois mots.

8 - A moins que l'une des fonctions membre affiche ou deplace ait été redéfinie dans pointcol.

5.6 Récapitulation

Voici un tableau récapitulant les propriétés des différentes sortes de dérivation (la mention "Accès FMA" signifie : accès aux fonctions membre ou amies de la classe ; la mention "nouveau statut" signifie : statut qu'aura ce membre dans une éventuelle classe dérivée).

Classe de base			Dérivée publique		Dérivée protégée		Dérivée privée	
Statut initial	Accès FMA	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur
public	O	O	public	O	protégé	N	privé	N
protégé	O	N	protégé	N	protégé	N	privé	N
privé	O	N	privé	N	privé	N	privé	N

Les différentes sortes de dérivation

Remarque :

On voit clairement qu'une dérivation protégée ne se distingue d'une dérivation privée que lorsque l'on est amené à dériver de nouvelles classes de la classe dérivée en question.

6. COMPATIBILITÉ ENTRE OBJETS D'UNE CLASSE DE BASE ET OBJETS D'UNE CLASSE DÉRIVÉE

D'une manière générale, en P.O.O., on considère qu'un objet d'une classe dérivée peut "remplacer" un objet d'une classe de base ou encore que : là où un objet de classe A est attendu, tout objet d'une classe dérivée de A peut "faire l'affaire".

Cette idée repose sur le fait que tout ce que l'on trouve dans une classe de base (fonctions ou données) se trouve également dans la classe dérivée. De même, toute action réalisable sur une classe de base peut toujours être réalisée sur une classe dérivée (ce qui ne veut pas dire pour autant que le résultat sera aussi satisfaisant dans le cas de la classe dérivée que dans celui de la classe de base – on affirme seulement qu'elle est possible !). Par exemple, un point coloré peut toujours être traité comme un point : il possède des coordonnées ; on peut afficher ces dernières comme on le ferait pour celles d'un point.

Bien entendu, les réciproques de ces deux propositions sont fausses ; par exemple, on ne peut pas colorer un point ou s'intéresser à sa couleur.

Cette compatibilité entre une classe dérivée et sa classe de base[9] va se retrouver en C++, avec une légère nuance : elle ne s'appliquera que dans le cas de dérivation publique[10]. Concrètement, cette compatibilité se résume à l'existence de conversions implicites :

- d'un objet d'un type dérivé dans un objet d'un type de base,
- d'un pointeur (ou d'une référence) sur une classe dérivée en un pointeur (ou une référence) sur une classe de base.

---

9 - Ou l'une de ses classes de base dans le cas de l'héritage multiple que nous aborderons dans le chapitre suivant.

10 - Ce qui se justifie par le fait que, dans le cas contraire, il suffirait de convertir un objet d'une classe dérivée dans le type de sa classe de base pour passer outre la privatisation des membres publics du type de base.

Nous allons voir ici l'incidence de ces conversions dans le cas d'affectations entre objets d'abord, entre pointeurs ensuite. La dernière situation, au demeurant la plus répandue, nous permettra de mettre en évidence :

- le typage statique des objets qui en découle ; ce point constituera en fait une introduction à la notion de méthode virtuelle permettant le typage dynamique (et qui fera l'objet du chapitre XV).
- les risques de violation du principe d'encapsulation qui en découlent.

6.1 Conversions d'un objet dérivé dans un objet d'un type de base

Soit nos deux classes "habituelles" :

```
class point                          class pointcol : public point
{     .....                          {     .....
```

Avec les déclarations :

```
point a ;
pointcol b ;
```

l'affectation :

```
a = b ;
```

est légale. Elle entraîne une conversion de b dans le type point[11] et l'affectation du résultat à a ; cette affectation se fait, suivant les cas :

- par appel de l'opérateur d'affectation (de la classe point) si celui-ci a été surdéfini,
- par emploi de l'affectation par défaut, dans le cas contraire.

En revanche, l'affectation suivante serait rejetée :

```
b = a ;
```

6.2 Conversions d'un pointeur sur une classe dérivée en un pointeur sur une classe de base

Considérons à nouveau une classe point et une classe pointcol dérivée de point, comportant chacune une fonction membre affiche.

```
class point                          class pointcol : public point
{     int x, y ;                     {     short couleur ;
  public :                             public :
      .....                                .....
      void affiche () ;                    void affiche () ;
      .....                                .....
} ;                                  } ;
```

Soit ces déclarations :

```
point * adp ;
```

11 - En général, une telle conversion est assez fictive et n'entraîne pas la création d'un nouvel objet de type point.

```
pointcol * adpc ;
```

Là encore, C++ autorise l'affectation :

```
adp = adpc ;
```

laquelle correspond à une conversion du type pointcol * dans le type point *.

L'affectation inverse :

```
adpc = adp ;
```

serait naturellement rejetée. Elle est cependant réalisable, en faisant appel à l'opérateur de "cast". Ainsi, bien que sa signification soit ici discutable[12], il vous sera toujours possible d'écrire l'instruction :

```
adpc = (pointcol *) adp ;
```

Remarque importante :

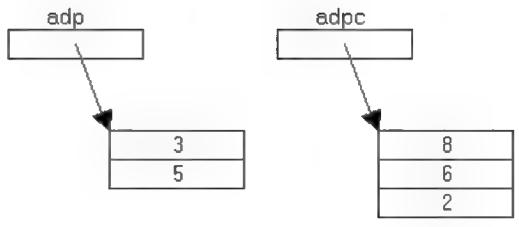
S'il est possible de convertir explicitement un pointeur de type point * en un pointeur de type pointcol *, il n'est, par contre, pas possible de convertir un objet de type pointcol en un objet de type point. La différence vient de ce que l'on a affaire à une "conversion prédéfinie" dans le premier cas[13], alors que dans le second cas, le compilateur ne peut "imaginer" ce que vous souhaitez faire.

6.3 Limitations liées au "typage statique" des objets

Considérons à nouveau les déclarations du paragraphe précédent accompagnées de[14] :

```
point p (3, 5) ; pointcol pc (8, 6, 2) ;
adp = & p      ; adpc = & pc ;
```

La situation est alors celle-ci :



A ce niveau, l'instruction :

```
adp -> affiche () ;
```

appellera la méthode point::affiche, tandis que l'instruction :

---

12 - Et même dangereuse, comme nous le verrons dans le paragraphe 6.4.

13 - Laquelle se borne en fait à un changement de type (sur le plan syntaxique), accompagné éventuellement d'un alignement d'adresse (attention, rien ne garantit que l'application successive des deux conversions réciproques (point * -> pointcol * puis pointcol * -> point *) fournisse exactement l'adresse initiale !).

14 - En supposant qu'il existe des constructeurs appropriés.

```
adpc -> affiche () ;
```

appellera la méthode pointcol::affiche.

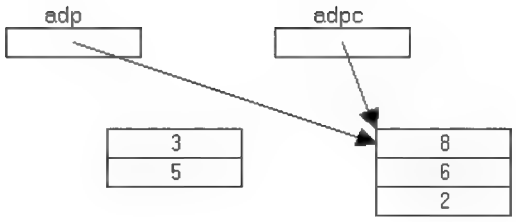
Nous aurions obtenu les mêmes résultats avec :

```
p.affiche ()  ;
pc.affiche () ;
```

Si nous exécutons alors l'affectation :

```
adp = adpc ;
```

nous aboutissons à cette situation :



Que va faire, à ce niveau, une instruction telle que :

```
adp -> affiche () ;
```

Y aura-t-il appel de point::affiche ou de pointcol::affiche ?

En effet, adp est du type point mais l'objet pointé par adp est du type pointcol. En fait, le choix de la méthode à appeler est réalisé par le compilateur, ce qui signifie que la méthode appelée est définie une fois pour toutes et qu'elle ne pourra pas évoluer au fil des changements éventuels de type de l'objet pointé. Bien entendu, dans ces conditions, on comprend que le compilateur ne peut que décider de mettre en place l'appel de la méthode correspondant au type défini par le pointeur. Ici, il s'agira donc de point::affiche, puisque adp est du type point *.

Notez bien que si pointcol dispose d'une méthode colore (n'existant pas dans point) un appel tel que :

```
adp -> colore (8) ;
```

sera rejeté par le compilateur.

On peut donc dire que, pour l'instant, le type des objets pointés par adp et adpc est décidé et figé au moment de la compilation. On peut alors considérer comme un leurre le fait que C++ tolère certaines conversions de pointeurs ; en fait, d'ailleurs, il tolère celles qui, au bout du compte, ne poseront pas de problème vis-à-vis du choix fait au moment de la compilation (comme nous l'avons dit, on pourra toujours afficher un pointcol "comme s'il" s'agissait d'un point). En effet, nous pouvons désigner, à l'aide d'un même pointeur, des objets de type différent, mais nous n'avons (pour l'instant) aucun moyen de tenir réellement compte du type de l'objet pointé (par exemple affiche traite un pointcol comme un point, mais elle ne peut pas savoir s'il s'agit d'un point ou d'un pointcol).

En réalité, nous verrons que C++ permet d'effectuer cette identification d'un objet au moment de l'exécution (et non plus, arbitrairement, à la compilation) et de réaliser ce que l'on nomme du "typage

dynamique" (alors que, jusqu'ici, nous n'avions affaire qu'à du typage "statique"). Ceci nécessitera l'emploi des **fonctions virtuelles** que nous aborderons dans le chapitre XV.

Voici maintenant un exemple de programme illustrant les limitations que nous venons d'évoquer ; remarquez que, dans la méthode affiche de pointcol, nous n'avons pas fait appel à la méthode affiche de point ; pour qu'elle puisse accéder aux membres x et y de point, nous avons prévu de leur donner le statut protégé.

```
#include <iostream.h>
class point
{ protected :            // pour que x et y soient accesibles à pointcol
   int x, y ;
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   void affiche ()
     { cout << "Je suis un point \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y << "\n" ;
     }
} ;
class pointcol : public point
{
   short couleur ;
  public :
   pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
     { couleur = cl ;
     }
   void affiche ()
     { cout << "Je suis un point coloré \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y ;
       cout << "   et ma couleur est :    " << couleur << "\n" ;
     }
} ;
main()
{  point p(3,5) ; point * adp = &p ;
   pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
   adp->affiche () ; adpc->affiche () ;
   cout << "-----------------\n" ;
   adp = adpc ;             //  adpc = adp serait rejeté
   adp->affiche () ; adpc->affiche () ;
}
Je suis un point
   mes coordonnées sont : 3 5
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
-----------------
Je suis un point
   mes coordonnées sont : 8 6
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
```

6.4 Les risques de violation des protections de la classe de base

N.B. Ce paragraphe peut être ignoré dans un premier temps.

Nous avons vu qu'il est possible, dans une classe dérivée, de rendre privés les membres publics de la classe de base. En voici un exemple :

```
class A                          class B : private A
{     int x ;                          {     int u ;
  public :                         public :
      float z ;                        double v ;
      void fa () ;                           void fb () ;
      .....                                  .....
} ;                              } ;
A a ;                            B b ;
```

Ici, l'objet a aura accès aux membres z et fa de A. On pourra écrire par exemple :

```
a.z = 5.25 ;
a.fa () ;
```

Par contre, l'objet b n'aura pas accès à ces membres, compte tenu du mot private figurant dans la déclaration de la classe B. Dans ces conditions, les instructions :

```
b.z = 8.3 ;
b.fa () ;
```

seront rejetées par le compilateur (à moins, bien sûr, que les membres z et fa ne soient redéfinis dans la classe B).

Néanmoins, il existe, pour l'utilisateur de la classe B un moyen de passer outre cette protection mise en place par le concepteur de la classe. En effet, il lui suffit de procéder ainsi :

```
A * ada ; B * adb ;
adb = &b ;
ada = (A *) adb ;
```

ou encore, plus brièvement :

```
A * ada = (A *) & b ;
```

Dans ces conditions, ada contient effectivement l'adresse de b (nous avons dû employer le "cast" car n'oubliez pas que, sinon, la conversion dérivée -> base serait rejetée) ; mais ada a toujours le type A *. On peut donc maintenant accéder aux membres publics de la classe A, bien que privés pour la classe B. Ces instructions seront acceptées :

```
ada -> z = 8.3 ;
ada -> fa () ;
```

7. CAS DU CONSTRUCTEUR DE RECOPIE

Rappelons que le constructeur de recopie (qu'il s'agisse de celui par défaut ou de celui qui est fourni explicitement) est appelé en cas :

- d'initialisation d'un objet par un objet de même type,
- de transmission de la valeur d'un objet en argument ou en retour d'une fonction.

Les règles que nous avons énoncées dans le paragraphe 4 s'appliquent à tous les constructeurs, et donc également au constructeur de recopie. Toutefois, dans ce dernier cas, il faut aussi tenir compte de l'existence d'un constructeur de recopie par défaut. Voyons plus précisément ce qu'il en est, en examinant les diverses situations possibles. Nous supposerons, pour fixer les idées, que l'on a affaire aux instructions suivantes (B dérive donc de A) :

```
class A { ... } ;
class B : public A { ... } ;
void fct (B) ; // fct est une fonction recevant un argument de type B
  ...
B b1 (...) ;   // arguments éventuels pour un "constructeur usuel"
fct (b1) ;     // appel de fct à qui on doit transmettre b1 par valeur, ce qui
               //   implique l'appel d'un constructeur de recopie de la classe
B
```

Bien entendu, tout ce que nous allons dire s'appliquerait également aux autres situations d'initialisation par recopie c'est-à-dire au cas où une fonction renverrait par valeur un résultat de type B ou encore à celui où l'on initialiserait un objet de type B avec un autre objet de type B, comme dans B b2 = b1 ou encore B b2 (b1) (revoyez éventuellement les paragraphes 3 et 4 du chapitre VII).

7.1 La classe dérivée (B) n'a pas défini de constructeur de recopie

Il y a donc appel du constructeur de recopie par défaut de B. Rappelons que la recopie se fait membre par membre. Nous avons vu ce que cela signifiait dans le cas des "objets membre". Ici, cela signifie que la "partie" de b1 appartenant à la classe A sera traitée comme un membre de type A. Il y aura donc appel du constructeur de recopie de A pour les membres donnée correspondants. Notez bien qu'alors, **si A a défini un tel constructeur**, il **sera appelé** ; dans le cas contraire, on se servira du constructeur de recopie par défaut de A[15].

Remarque importante :

Cette généralisation de la recopie membre par membre au cas des classes dérivées pourrait laisser supposer qu'il en ira de même pour l'opérateur d'affectation (dont nous avons vu qu'il fonctionnait de façon semblable à la recopie). En fait, ce ne sera pas le cas ; nous y reviendrons dans le paragraphe 9.

7.2 La classe dérivée (B) a défini un constructeur de recopie

Dans ce cas, le constructeur de recopie de B est naturellement appelé. La question qui se pose alors est de savoir s'il y a appel d'un constructeur de A. En fait, C++ a décidé de ne prévoir aucun appel automatique de constructeur de la classe de base dans ce cas (même s'il existe un constructeur de recopie dans A !). Cela

15 - Dans les versions antérieures à la 2.0, on avait affaire à une recopie globale.

signifie donc que le constructeur de recopie de la classe dérivée doit prendre en charge l'intégralité de la recopie de l'objet et non seulement sa partie héritée.

Mais, il reste possible d'utiliser le mécanisme de transmission d'informations entre constructeurs (étudiée dans le paragraphe 4.3). Ainsi, si le constructeur de B prévoit des informations pour un constructeur de A avec un en-tête de la forme :

```
B (B & x) : A (...)
```

il y aura appel du constructeur correspondant de A.

En général, on souhaitera que le constructeur de A appelé à ce niveau soit le constructeur de recopie de A[16]. Dans ces conditions, on voit que ce constructeur doit recevoir en argument, non pas l'objet x tout entier, mais seulement ce qui, dans x, est de type A. C'est là qu'intervient précisément la possibilité de conversion implicite d'une classe dérivée dans une classe de base (étudiée dans le paragraphe 6). Il nous suffira de définir ainsi notre constructeur pour aboutir à une recopie satisfaisante :

```
B (B & x) : A (x)   // x, de type B, est converti dans le type A pour être
transmis
                    //   au constructeur de recopie de A
{ // recopie de la partie de x spécifique à B (non héritée de A)
}
```

Voici un programme illustrant cette possibilité : nous y définissons simplement nos deux classes habituelles point et pointcol en les munissant toutes les deux d'un constructeur de recopie[17] et nous provoquons l'appel de celui de pointcol en appelant une fonction fct à un argument de type pointcol transmis par valeur.

```
#include <iostream.h>
class point
{  int x, y ;
  public :
   point (int abs=0, int ord=0)         // constructeur usuel
     { x = abs ; y = ord ;
       cout << "++ point    " << x << " " << y << "\n" ;
     }
   point (point & p)                    // constructeur de recopie
     { x = p.x ; y = p.y ;
       cout << "CR point    " << x << " " << y << "\n" ;
     }
} ;
class pointcol : public point
{  char coul ;
  public :
```

16 - Bien entendu, en théorie, il reste possible au constructeur par recopie de la classe dérivée B d'appeler n'importe quel constructeur de A, autre que son constructeur par recopie. Dans ce cas, il faut être en mesure de reporter convenablement dans l'objet les valeurs de la partie de x qui est un A ; dans certains cas, on pourra encore y parvenir par le mécanisme de transmission d'informations entre constructeurs ; dans le cas contraire, il faudra effectuer le travail au sein du constructeur de recopie de B, ce qui peut s'avérer délicat, compte tenu d'éventuels problèmes de droits d'accès...

17 - Ce qui, dans ce cas précis de classe ne comportant pas de pointeurs, n'est pas utile, la recopie par défaut décrite précédemment s'avérant suffisante dans tous les cas. Mais, ici, l'objectif est simplement d'illuster le mécanisme de transmission d'informations entre constructeurs.

```
   pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord) // constructeur
usuel
     { coul = cl ;
       cout << "++ pointcol " << int(coul) << "\n" ;
     }
   pointcol (pointcol & p) : point (p)  // constructeur de recopie
                                        // il y aura conversion implicite
                                        // de p dans le type point
     { coul = p.coul ;
       cout << "CR pointcol " << int(coul) << "\n" ;
     }
} ;
void fct (pointcol pc)
{
   cout << "*** entrée dans fct ***\n" ;
}

main()
{  void fct (pointcol) ;
   pointcol a (2,3,4) ;
   fct (a) ;                // appel de fct, à qui on transmet a par valeur
}
```

```
++ point    2 3
++ pointcol 4
CR point    2 3
CR pointcol 4
*** entrée dans fct ***
```

Pour forcer l'appel d'un constructeur de recopie de la classe de base

8. OPÉRATEUR D'AFFECTATION ET HÉRITAGE

Nous avons déjà vu comment C++ définit l'affectation par défaut entre deux objets de même type. D'autre part, nous avons vu comment il était possible de surdéfinir cet opérateur d'affectation (obligatoirement sous forme d'une fonction membre).

Voyons ce que deviennent ces possibilités en cas d'héritage. Supposons que la classe B hérite (publiquement) de A et considérons, comme nous l'avons fait pour le constructeur de recopie (paragraphe 4.6) les différentes situations possibles.

8.1 La classe dérivée (B) n'a pas surdéfini l'opérateur =

L'affectation de deux objets de type B se déroule membre à membre en considérant que la "partie héritée de A" constitue un membre. Ainsi, les membres propres à B sont traités par l'affectation prévue pour leur type (par défaut ou surdéfinie, suivant le cas). La partie héritée de A est traitée par l'affectation prévue dans la classe A, c'est-à-dire par l'opérateur = surdéfini dans A s'il existe, par l'affectation par défaut dans le cas contraire.

On retrouve un comportement tout à fait analogue à celui décrit dans le cas du constructeur de recopie.

8.2 La classe dérivée (B) a surdéfini l'opérateur =

Dans ce cas, l'affectation de deux objets de type B fera nécessairement appel à l'opérateur = défini dans B. Celui de A ne sera pas appelé, même s'il a été surdéfini. Il **faudra donc que l'opérateur = de B prenne en charge tout ce qui concerne l'affectation d'objets de type B**, y compris pour ce qui est des membres hérités de A.

Voici un premier exemple de programme qui illustre cela : la classe pointcol dérive de point. Les deux classes ont surdéfini l'opérateur =.

```
#include <iostream.h>
class point
{  protected :
     int x, y ;
   public :
     point (int abs=0, int ord=0) { x=abs ; y=ord ;}
     point & operator = (point & a)
      { x = a.x ; y = a.y ;
        cout << "opérateur = de point \n" ;
        return * this ;
      }
} ;

class pointcol : public point
{  protected :
     int coul ;
   public :
     pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord) { coul=cl ; }
     pointcol & operator = (pointcol & b)
      { coul = b.coul ;
        cout << "opérateur = de pointcol\n" ;
        return * this ;
      }
     void affiche () { cout << "pointcol : " << x << " " << y << " " << coul <<
"\n" ;
                     }
} ;
main()
{ pointcol p(1, 3, 10) , q(4, 9, 20) ;
  cout << "p       = " ; p.affiche () ;
  cout << "q avant = " ; q.affiche () ;
  q = p ;
  cout << "q après = " ; q.affiche () ;
}
```

```
p       = pointcol : 1 3 10
q avant = pointcol : 4 9 20
```

```
opérateur = de pointcol
q après = pointcol : 4 9 10
```

Quand la classe de base et la classe dérivée surdéfinissent l'opérateur =

On y voit clairement que l'opérateur = défini dans la classe point n'a pas été appelé lors d'une affectation entre objets de type pointcol.

Le problème est voisin de celui rencontré à propos du constructeur de recopie, avec cette différence qu'on ne dispose plus ici du mécanisme de transfert d'arguments qui en permettait un appel (presque) implicite. Ici, si l'on veut pouvoir profiter de l'opérateur = défini dans A, il faudra l'appeler explicitement. Le plus simple pour ce faire est d'utiliser les possibilités de conversions de pointeurs examinées dans le précédent paragraphe.

Voici, par exemple, comment nous pourrions modifier dans ce sens l'opérateur = de pointcol :

```
      pointcol & operator = (pointcol & b)
       { point * ad1, * ad2 ;
         cout << "opérateur = de pointcol\n" ;
         ad1 = this ;     // conversion pointeur sur pointcol en pointeur sur
point
         ad2 = & b ;      //   idem
         * ad1 = * ad2 ; // affectation de la "partie point" de b
         coul = b.coul ; // affectation de la partie propre à pointcol
         return * this ;
       }
```

Nous convertissons les pointeurs (this et &b) sur des objets de pointcol en des pointeurs sur des objets de type point. Il suffit ensuite de réaliser une affectation entre les nouveaux objets pointés (* ad1 et * ad2) pour entraîner l'appel de l'opérateur = de la classe point. Voici le nouveau programme complet ainsi modifié. Cette fois, les résultats montrent que l'affectation entre objets de type pointcol est satisfaisante.

```
#include <iostream.h>
class point
{ protected :
     int x, y ;
   public :
     point (int abs=0, int ord=0) { x=abs ; y=ord ;}
     point & operator = (point & a)
      {  x = a.x ; y = a.y ;
         cout << "opérateur = de point \n" ;
         return * this ;
      }
} ;
class pointcol : public point
{  protected :
     int coul ;
   public :
     pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord) { coul=cl ; }
```

```
      pointcol & operator = (pointcol & b)
       { point * ad1, * ad2 ;
         cout << "opérateur = de pointcol\n" ;
         ad1 = this ;    // conversion pointeur sur pointcol en pointeur sur
point
         ad2 = & b ;     //   idem
         * ad1 = * ad2 ; // affectation de la "partie point" de b
         coul = b.coul ; // affectation de la partie propre à pointcol
         return * this ;
       }
      void affiche ()
       { cout << "pointcol : " << x << " " << y << " " << coul << "\n" ;
       }
} ;
main()
{ pointcol p(1, 3, 10) , q(4, 9, 20) ;
  cout << "p       = " ; p.affiche () ;
  cout << "q avant = " ; q.affiche () ;
  q = p ;
  cout << "q après = " ; q.affiche () ;
}
```

```
p       = pointcol : 1 3 10
q avant = pointcol : 4 9 20
opérateur = de pointcol
opérateur = de point
q après = pointcol : 1 3 10
```

Pour forcer, dans une classe dérivée, l'utilisation de l'opérateur = surdéfini dans la classe de base

Remarque :

On dit souvent qu'en C++, l'opérateur d'affectation **n'est pas hérité**. Une telle affirmation est en fait source de confusions. En effet, on peut considérer qu'elle est exacte, dans la mesure où, lorsque B n'a pas défini l'opérateur =, on ne se contente pas de faire appel à celui défini (éventuellement) dans A (ce qui reviendrait à réaliser une affectation partielle ne concernant que la partie héritée de A !). En revanche, on peut considérer que cette affirmation est fausse puisque, lorsque B ne surdéfinit pas l'opérateur =, cette classe peut quand même "profiter" (automatiquement) de l'opérateur défini dans A;

Cette ambiguïté se retrouve d'ailleurs dans certaines implémentations qui, suivant cette affirmation à la lettre, n'ont pas prévu d'appeler l'opérateur = défini dans A lorsque B n'en définissait aucun. Le résultat est alors assez déconcertant...

9. HÉRITAGE ET FORME CANONIQUE D'UNE CLASSE

Dans le paragraphe 4 du chapitre IX, nous avons défini ce que l'on nomme la "forme canonique" d'une classe, c'est-à-dire le canevas suivant lequel devrait être construite toute classe non triviale (disposant de pointeurs).

Voici comment ce schéma pourrait être généralisé dans le cadre de l'héritage (par souci de brièveté, certaines fonctions ont été placées "en ligne"). Plus précisément, nous y trouvons :

- une classe de base nommée T, respectant la forme canonique déjà présentée,
- une classe dérivée nommée U, respectant elle aussi la forme canonique, mais s'appuyant sur certaines des fonctionnalités de sa classe de base (constructeur par recopie et opérateur d'affectation). Notez bien que nous ne faisons que récapituler ici ce qui a été présenté dans les paragraphes 7 et 8 de ce chapitre.

```
class T
{ public :
   T (...) ;                        // constructeurs de T, autres que par recopie
   T (const T &) ;                  // constructeur de recopie de T (forme
conseillée)
   ~T () ;                             // destructeur
   T & T::operator = (const T &) ;     // opérateur d'affectation (forme
conseillée)
   .....
} ;
class U : public T
{ public :
   U (...) ;                   // constructeurs autres que recopie
   U (const U & x) : T (x)     // constructeur de recopie de U : il utilise
celui de T
      {
        // prévoir ici la copie de la partie de x spécifique à T (qui n'est pas
un T)
      }
   ~U () ;
   U & U::operator = (const U & x)    // opérateur d'affectation (forme
conseillée)
     {  T * ad1 = this, * ad2 = &x ;
        *ad1 = *ad2 ;                 // affectation (à l'objet courant) de la
partie
                                      //   de x héritée de T
        // prévoir ici l'affectation (à l'objet courant)
        //  de la partie de x spécifique à U (non héritée de T)
     }
```

Forme canonique d'une classe dérivée

10. CE QU'EST L'HÉRITAGE ET CE QU'IL N'EST PAS

Nous avons vu comment une classe dérivée peut tirer parti des possibilités d'une classe de base. On traduit parfois cela en disant qu'elle hérite de ses "fonctionnalités". Ce dernier terme peut, cependant, prêter à confusion en laissant croire que l'héritage est plus général qu'il ne l'est en réalité.

Prenons l'exemple d'une classe point qui a surdéfini l'opérateur + (point + point -> point) et d'une classe pointcol qui hérite publiquement de point (et qui ne redéfinit pas +). Pourra-t-elle utiliser cette "fonctionnalité" de la classe point qu'est l'opérateur + ? En fait, on voit à ce niveau qu'un certain nombre

de choses sont floues. La somme de deux points colorés sera-t-elle un point coloré ? Si oui, quelle pourrait bien être sa couleur ? Sera-t-elle simplement un point ? Dans ce cas, on ne peut pas vraiment dire que pointcol a hérité des possibilités d'addition de point.

Prenons maintenant un autre exemple : celui de la classe point, munie d'une fonction (membre ou amie) coincide, telle que nous l'avions considérée dans le paragraphe 1 du chapitre VIII et une classe pointcol héritant de point. Notre fonction coincide pourra-t-elle (telle qu'elle est) être utilisée pour tester la coïncidence de deux points colorés ?

Nous vous proposons d'apporter des éléments de réponse à ces différentes questions. Pour ce faire, nous allons préciser exactement ce qu'est l'héritage, ce qui nous permettra de conclure que les situations que nous venons d'évoquer ne relèvent pas (uniquement) de l'héritage. Nous verrons ensuite comment la conjugaison de l'héritage et des règles de compatibilité entre objets dérivés (dont nous avons parlé ci-dessus) permet de donner un sens à certaines des situations évoquées ; les autres nécessiteront le recours à des moyens supplémentaires (conversions, par exemple).

10.1 La situation d'héritage

Considérons ce canevas (t désignant un type quelconque) :

```
class A                              class B : public A
{     .....                                {     .....
  public :                           } ;
      t f(.....) ;
      .....
} ;
```

La classe A possède une fonction membre f, dont nous ne précisons pas ici les arguments, fournissant un résultat de type t (type de base ou défini par l'utilisateur). La classe B hérite des membres publics de A, donc de f. Soit deux objets a et b :

```
A a ; B b ;
```

Bien entendu, l'appel :

```
a.f (.....) ;
```

a un sens et il fournit un résultat de type t.

Le fait que B hérite publiquement de A permet alors d'affirmer que l'**appel** :

```
b.f (.....) ;
```

a lui aussi **un sens**, autrement dit, que f agira sur b (ou avec b) comme s'il était du type A. **Son résultat sera toujours de type t** et ses arguments auront toujours le type imposé par son prototype.

Tout l'héritage est contenu dans cette affirmation à laquelle il faut absolument se tenir. Expliquons-nous.

a) Le type du résultat de l'appel

Généralement, tant que t est un type usuel, l'affirmation ci-dessus apparaît triviale. Mais des doutes apparaissent dès que t est un type objet, surtout s'il s'agit du type de la classe dont f est membre. Ainsi, avec le prototype :

```
A f(.....)
```

le résultat de l'appel b.f(.....) sera bien de type A (et non de type B comme on pourrait parfois le souhaiter...).

b) Le type des arguments de f

La remarque faite à propos de la valeur de retour s'applique aux arguments de f. Par exemple, supposons que f ait pour prototype :

```
t f(A) ;
```

et que nous ayons déclaré :

```
A a1, a2 ; B b1 b2 ;
```

L'héritage (public) donne effectivement une signification à :

```
b1.f(a1)
```

Quant à l'appel :

```
b1.f(b2)
```

s'il a un sens, c'est grâce à l'existence de conversions implicites :

- de l'objet b1 de type B en un objet du type A si f reçoit son argument par valeur. N'oubliez pas qu'alors il y aura appel d'un constructeur de recopie (par défaut ou surdéfini),
- d'une référence à b1 de type B en une référence à un objet de type A si f reçoit ses arguments par référence.

10.2 Exemples

Revenons maintenant aux exemples évoqués en introduction de ce paragraphe.

a) Héritage dans pointcol d'un opérateur + défini dans point

En fait, que l'opérateur + soit défini sous forme d'une fonction membre ou d'une fonction amie, la "somme" de deux objets a et b de type pointcol sera de type point. En effet, dans le premier cas, l'expression :

```
a + b
```

sera évaluée comme :

```
a.operator+ (b)
```

Il y aura appel de la fonction membre operator+[18] pour l'objet a (dont on ne considérera que ce qui est du type point), à laquelle on transmettra en argument le résultat de la conversion de b en un point[19]. Son résultat sera de type point.

---

18 - Fonction membre de pointcol, mais héritée de point.

Dans le second cas, l'expression sera évaluée comme :

```
operator+ (a, b)
```

Il y aura appel de la fonction amie[20] operator+ , à laquelle on transmettra le résultat de la conversion de a et b dans le type point. Le résultat sera toujours de type point.

Dans ces conditions, vous voyez que si c est de type pointcol, une banale affectation telle que :

```
c = a + b ;
```

sera rejetée, faute de disposer de la conversion de point en pointcol. On peut d'ailleurs logiquement se demander quelle couleur une telle conversion pourrait attribuer à son résultat ? Si maintenant on souhaite définir la somme de deux points colorés, il faudra redéfinir l'opérateur + au sein de pointcol, quitte à ce qu'il fasse appel à celui défini dans point pour la somme des coordonnées.

b) Héritage dans pointcol de la fonction coincide de point

Cette fois, il est facile de voir qu'aucun problème particulier ne se pose[21], à partir du moment où l'on considère que la coïncidence de deux points colorés correspond à l'égalité de leurs seules coordonnées (la couleur n'intervenant pas).

A titre indicatif, voici un exemple de programme complet, dans lequel coincide est défini comme une fonction membre de point.

```
#include <iostream.h>
class point
{
   int x, y ;
 public :
  point (int abs=0, int ord=0)  { x=abs ; y=ord ; }
  friend int coincide (point &, point &) ;
} ;

int coincide (point & p, point & q)
{   if ((p.x == q.x) && (p.y == q.y)) return 1 ;
                                 else return 0 ;
}
class pointcol : public point
{ short couleur ;
 public :
  pointcol (int abs=0, int ord=0, short cl=1) : (abs, ord)
   { couleur = cl ; }
} ;

main()                                          // programme d'essai
{
```

19 - Suivant les cas, il y aura conversion d'objets ou conversion de références.

20 - Amie de point et de pointcol par héritage, mais, ici, c'est seulement la relation d'amitié avec point qui est employée.

21 - Mais, ici, le résultat fourni par coincide n'est pas d'un type classe !

```
    pointcol a(2,5,3), b(2,5,9), c ;
    if (coincide (a,b)) cout << "a coincide avec b \n" ;
                   else cout << "a et b sont différents \n" ;
    if (coincide (a,c)) cout << "a coincide avec c \n" ;
                   else cout << "a et c sont différents \n" ;
}
```

```
a coincide avec b
a et c sont différents
```

Héritage, dans pointcol, de la fonction coincide de point

11. EXEMPLE DE CLASSE DÉRIVÉE

Supposez que nous disposions de la classe vect telle que nous l'avions définie dans le paragraphe 5 du chapitre IX. Cette classe est munie d'un constructeur, d'un destructeur et d'un opérateur d'indiçage [] (notez bien que, pour être exploitable, cette classe qui contient des parties dynamiques, devrait comporter également un constructeur par recopie et la surdéfinition de l'opérateur d'affectation).

```
class vect
{  int nelem ;
   int * adr ;
 public :
   vect (int n) { adr = new int [nelem=n] ; }
   ~vect () {delete adr ;}
   int & operator [] (int) ;
} ;
int & vect::operator [] (int i)
{ return adr[i] ; }
```

Supposez que nous ayons besoin de vecteurs dans lesquels on puisse fixer, non seulement le nombre d'éléments, mais les bornes (minimum et maximum) des indices (supposés toujours entiers). Par exemple, nous pourrions déclarer (si vect1 est le nom de la nouvelle classe) :

```
vect1 t (15, 24) ;
```

Ce qui signifierait que t est un tableau de dix entiers d'indices variant de 15 à 24.

Il semble alors naturel d'essayer de dériver une classe de vect. Il nous faut prévoir deux membres supplémentaires pour conserver les bornes de l'indice, d'où le début de la déclaration de notre nouvelle classe :

```
class vect1 : public vect
{     int debut, fin ;
```

Manifestement, vect1 nécessite un constructeur à deux arguments entiers correspondant aux bornes de l'indice. Son en-tête sera de la forme :

```
vect1 (int d, int f)
```

Mais l'appel de ce constructeur entraînera automatiquement celui du constructeur de vect ; il n'est donc pas question de faire dans vect1 l'allocation dynamique de notre vecteur ; au contraire, nous réutilisons le travail effectué par vect : il nous suffit de lui transmettre le nombre d'éléments souhaités, d'où l'en-tête complet de vect1 :

```
vect1 (int d, int f) : vect (f-d+1)
```

Quant à la tâche spécifique de vect1, elle se limite à renseigner les valeurs de début et fin.

A priori, la classe vect1 n'a pas besoin de destructeur, dans la mesure où elle n'alloue aucun emplacement dynamique autre que celui déjà alloué par vect.

Nous pourrions aussi penser que vect1 n'a pas besoin de surdéfinir l'opérateur [], dans la mesure où elle "hérite" de celui de vect. Qu'en est-il exactement ? Dans vect, la fonction membre operator[] reçoit un argument implicite et un argument de type int ; elle fournit une valeur de type int. Sur ce plan, donc, l'héritage va fonctionner correctement et C++ acceptera qu'on fasse appel à operator[] pour un objet de type dérivé vect1. Ainsi, avec :

```
vect1 t (15, 24)
```

la notation :

```
t[i]
```

qui signifiera

```
t.operator[] (i)
```

aura bien une signification.

Le seul ennui... c'est que cette notation désignera toujours le $i^{ème}$ élément du tableau dynamique de l'objet t. Et ce n'est plus ce que nous voulons. Il nous faut donc surdéfinir l'opérateur [] pour la classe vect1.

A ce niveau, (au moins) deux solutions s'offrent à nous :

- utiliser l'opérateur existant dans vect, ce qui nous conduit à :

```
int & operator[] (int i)
{     return vect::operator[] (i-debut) ; }
```

- ne pas utiliser l'opérateur existant dans vect, ce qui nous conduirait à :

```
int & operator [] (int i)
{     return adr [i-debut] ; }
```

(à condition que adr soit accessible à la fonction operator [], donc déclaré public ou, plus raisonnablement, privé).

Cette dernière solution paraît peut-être plus séduisante[22].

[22] - Du moins ici, car le travail à effectuer était simple. Dans la pratique, on cherchera plutôt à récupérer le travail déjà effectué, en se contentant de le compléter si nécessaire.

Voici un exemple complet faisant appel à la première solution. Nous y avons fait figurer la classe vect elle-même pour vous faciliter son examen et nous avons, comme à l'accoutumée, introduit quelques affichages d'information, au sein de certaines fonctions membre de vect et de vect1.

```
#include <iostream.h>
// ***************** la classe vect *********************************
class vect
{   int nelem ;                              // nombre d'éléments
    int * adr ;                              // pointeur sur ces éléments
  public :
    vect (int n)                             // constructeur vect
    { adr = new int [nelem = n] ;
      cout << "+ Constr. vect de taille " << n << "\n" ;
    }
    ~vect ()                                 // destructeur vect
    { cout << "- Destr. vect " ;
      delete adr ;
    }
    int & operator [] (int) ;
} ;
int & vect::operator [] (int i)
  { return adr[i] ;
  }
// ***************** la classe dérivée : vect1 **********************
class vect1 : public vect
{  int debut, fin ;
  public :
   vect1 (int d, int f) : vect (f - d + 1)   // constructeur vect1
     { cout << "++ Constr. vect1 - bornes : " << d << " " << f << "\n" ;
       debut = d ; fin = f ;
     }
   int & operator [] (int) ;
} ;
int & vect1::operator [] (int i)
  { return vect::operator [] (i-debut) ; }
// ***************** un programme d'essai ****************************
main()
{ const int MIN=15, MAX = 24 ;
  vect1 t(MIN, MAX)  ;
  int i ;
  for (i=MIN ; i<=MAX ; i++) t[i] = i ;
  for (i=MIN ; i<=MAX ; i++) cout << t[i] << " " ;
  cout << "\n" ;
}
```

```
+ Constr. vect de taille 10
++ Constr. vect1 - bornes : 15 24
15 16 17 18 19 20 21 22 23 24
- Destr. vect
```

Remarque :

Bien entendu, là encore, pour être exploitable, la classe vect1 devrait définir un constructeur par recopie et l'opérateur d'affectation. A ce propos, on peut noter qu'il reste possible de définir ces deux fonctions dans vect1, même si elles n'ont pas été définies correctement dans vect.

12. PATRONS DE CLASSES ET HÉRITAGE

Il est très facile de combiner la notion d'héritage avec celle de patron de classes. Cette combinaison peut revêtir plusieurs aspects :

- **Classe "ordinaire" dérivée d'une classe patron** (c'est-à-dire d'une instance particulière d'un patron de classe) ; par exemple, si A est une classe patron définie par template < class T> A :

  ```
  class B : public A <int>    // B dérive de la classe patron A<int>
  ```

  on obtient une seule classe nommée B.

- **Patron de classes dérivé d'une classe "ordinaire"**, par exemple (A étant une classe ordinaire) :

  ```
  template <class T> class B : public A
  ```

  On obtient une famille de classes (de paramètre de type T). L'aspect "patron" a été introduit ici au moment de la dérivation.

- **Patron de classes dérivé d'un patron de classes.** Cette possibilité peut revêtir deux aspects suivant que l'on introduit ou non de nouveaux paramètres lors de la dérivation. Par exemple, si A est une classe patron définie par template < class T> A, on peut :

  - définir une nouvelle famille de fonctions dérivées par :

    ```
    template <class T> class B : public A <T>
    ```

    Dans ce cas, il existe autant de classes dérivées possibles que de classes de base possibles.

  - définir une nouvelle famille de fonctions dérivées par :

    ```
    template <class T, class U> class B : public A <T>
    ```

    Dans ce cas, on peut dire que chaque classe de base possible peut engender une famille de classes dérivées (de paramètre de type U).

D'une manière générale, vous pouvez "jouer" à volonté avec les paramètres, à savoir en introduire ou en supprimer à volonté.

Voici trois exemples correspondant à certaines des situations que nous venons d'évoquer.

a) Classe "ordinaire" pointcol_int dérivant d'une classe patron point< int>

```
#include <iostream.h>
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)  {  x = abs ; y = ord ; }
   void affiche () {  cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;
class pointcol_int : public point <int>
{  int coul ;
  public :
   pointcol_int (int abs=0, int ord=0, int cl=1) : point <int> (abs, ord)
    { coul = cl ;
    }
   void affiche ()
    { point<int>::affiche () ; cout << "      couleur : " << coul << "\n" ;
    }
} ;
main ()
{  point <float> pf (3.5, 2.8) ; pf.affiche () ; // instanciation d'une classe
patron
   pointcol_int p (3, 5, 9) ; p.affiche ();      // emploi (classique) de la
classe
                                                 //      pointcol_int
}
```

```
Coordonnées : 3.5 2.8
Coordonnées : 3 5
      couleur : 9
```

b) Patron de classes pointcol dérivant d'un patron de classes point, avec les mêmes paramètres

A partir du patron template < class T> class point, nous dérivons un patron nommé pointcol dans lequel le nouveau membre introduit est du même type T que les coordonnées du point.

```
#include <iostream.h>
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)  {  x = abs ; y = ord ; }
   void affiche () {  cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;
template <class T> class pointcol : public point <T>
{  T coul ;
  public :
   pointcol (T abs=0, T ord=0, T cl=1) : point <T> (abs, ord) { coul = cl ; }
   void affiche () { point<T>::affiche () ; cout << "      couleur : " << coul
; }
} ;
```

```
main ()
{  point <long> p (34, 45) ; p.affiche () ;
   pointcol <short> q (12, 45, 5) ; q.affiche () ;
}
```

```
Coordonnées : 34 45
Coordonnées : 12 45
      couleur : 5
```

c) Patron de classes dérivant d'un patron de classes et introduisant un nouveau paramètre

A partir du patron template < class T> class point, nous dérivons un patron nommé pointcol dans lequel le nouveau membre introduit est d'un type U différent de celui des coordonnées du point.

```
#include <iostream.h>
template <class T> class point
{  T x ; T y ;
  public :
   point (T abs=0, T ord=0)  {  x = abs ; y = ord ; }
   void affiche () {  cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;
template <class T, class U> class pointcol : public point <T>
{  U coul ;
  public :
   pointcol (T abs=0, T ord=0, U cl=1) : point <T> (abs, ord) { coul = cl ; }
   void affiche ()
    { point<T>::affiche () ; cout << "      couleur : " << coul << "\n" ;
    }
} ;
main ()
{
          // un point à coordonnées de type float et couleur de type int
   pointcol <float, int> p (3.5, 2.8, 12) ; p.affiche () ;
          // un point à coordonnées de type unsigned long et couleur de type
short
   pointcol <unsigned long, short> q (295467, 345789, 8) ; q.affiche () ;
}
```

```
Coordonnées : 3.5 2.8
      couleur : 12
Coordonnées : 295467 345789
      couleur : 8
```

13. RETOUR SUR LES POINTEURS SUR DES FONCTIONS MEMBRE

N.B. Ce paragraphe peut être ignoré dans un premier temps.

Le paragraphe 9 du chapitre VI vous a montré comment déclarer et utiliser des pointeurs sur des fonctions membre. Voyons ce que devient cette notion dans le contexte de l'héritage. Considérons ces deux classes :

```
class point                          class pointcol : public point
{     .....                          {     .....
  public :                         public :
      void dep_hor (int) ;                 void colore (int) ;
      void dep_vert (int) ;                .....
      .....                          } ;
} ;
```

Nous pouvons déclarer un pointeur adfp sur des fonctions membre de point par :

```
void (point:: * adfp) (int) ;
```

Cela signifie que adfp peut recevoir l'adresse de n'importe quelle fonction membre de point, dans la mesure où son prototype est de la forme void f (int).

De même, nous pouvons déclarer :

```
void (pointcol:: * adfpc) (int) ;
```

Bien entendu, ces affectations sont légales :

```
adfp  = point::dep_hor ;
adfp  = point::dep_vert ;
adfpc = pointcol::colore ;
```

Il en va de même pour :

```
adfpc = pointcol::dep_hor ;
adfpc = pointcol::dep_vert ;
```

puisque les fonctions dep_hor et dep_vert sont effectivement (également) des membres de pointcol[23].

Mais on peut aussi s'interroger sur la "compatibilité" existant entre adfp et adfpc. Autrement dit, lequel peut être affecté à l'autre ? Nous donnerons un peu plus loin la règle adoptée par C++ mais, auparavant, nous allons vous montrer que celle-ci est logique.

Il suffit, en effet, de penser à l'usage qui est fait de tels pointeurs, à savoir l'appel de la fonction correspondante. Si l'on accepte que adfpc reçoive une valeur du type pointeur sur une fonction membre de point (de même prototype), cela signifie qu'on pourra être amené à appeler, pour un objet de type pointcol[24], une fonction héritée de point. Cela ne pose donc aucun problème. En revanche, si l'on acceptait que adfp reçoive une valeur du type pointeur sur une fonction membre de pointcol, cela signifierait qu'on pourrait être amené à appeler, pour un objet de type point, n'importe quelle fonction de pointcol.

---

[23] - Pour le compilateur, point::dep_hor et pointcol::dep_hor sont de type différent. Cela n'empêche pas ces deux symboles de désigner la même adresse.

[24] - Car, bien entendu, une affectation telle que adfpc = adfp ne modifie pas le type de adfpc.

Manifestement, certaines fonctions (celles définies dans pointcol, c'est-à-dire celles qui ne sont pas héritées de point) risqueraient de ne pas pouvoir travailler correctement !

D'où la règle prévue par C++ :

> **Il existe une conversion implicite d'un pointeur sur une fonction membre d'une classe dérivée en un pointeur sur une fonction membre (de même prototype) d'une classe de base.**

Remarque :

> Si on se limite aux apparences (c'est-à-dire si on ne cherche pas à comprendre les raisons profondes), cette règle semble "diverger" par rapport aux conversions implicites entre objets ou pointeurs sur des objets : ces dernières se font dans le sens dérivée -> base, alors que, dans le cas des fonctions membre, elles ont lieu dans le sens base -> dérivée.

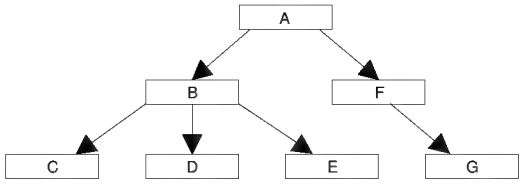
14. L'HÉRITAGE EN GÉNÉRAL

Nous venons de vous exposer les principes de base de l'héritage en nous limitant à des situations ne faisant intervenir que deux classes à la fois : une classe de base et une classe dérivée.

En fait, ces notions de classe de base et de classe dérivée sont relatives puisque :

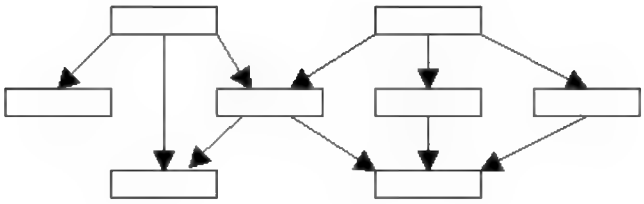
- d'une même classe peuvent être dérivées plusieurs classes différentes (éventuellement utilisées au sein d'un même programme),
- une classe dérivée peut, à son tour, servir de classe de base pour une autre classe dérivée.

Autrement dit, les différentes classes dérivées d'une même classe de base peuvent être représentées par une "arborescence" telle que la suivante :



Ici, C est dérivée de B, elle-même dérivée de A (on dit aussi que C hérite de B qui, elle-même hérite de A). Pour traduire la relation existant entre A et C, on dira que C est une descendante de A ou encore que A est une ascendante de C. Naturellement, C est aussi une descendante de B ; lorsqu'on aura besoin d'être plus précis, on dira que C est une descendante directe de B.

Par ailleurs, C++ (depuis la version 2) élargit les possibilités d'héritage en introduisant ce que l'on nomme l'**héritage multiple** : une classe donnée peut hériter simultanément de plusieurs classes. Dans ces conditions, on n'a plus affaire à une arborescence de classes, mais à un graphe qui peut éventuellement devenir complexe. En voici un simple exemple :



Tout ce qui a été dit jusqu'à maintenant se généralise sans aucun problème à toutes les situations d'héritage simple (syntaxe, appel des constructeurs...). D'une manière générale, lorsque nous parlerons d'une classe dérivée d'une classe de base, il pourra s'agir d'une descendante quelconque (directe ou non). De même, lorsque nous parlerons de dérivation publique, il faudra comprendre que la classe concernée s'obtient par une ou plusieurs dérivations successives publiques de sa classe de base. Notez **qu'il suffira qu'une seule de ces dérivations soit privée pour qu'au bout du compte, on parle globalement de dérivation privée.**

En ce qui concerne les situations d'héritage multiple, leur mise en œuvre nécessite quelques connaissances supplémentaires. Nous avons préféré les regrouper dans un chapitre séparé (XIV), compte tenu, notamment, de ce que leur usage est peu répandu.

15. EXPLOITATION D'UNE CLASSE DÉRIVÉE

L'héritage peut être utilisé dans des buts très différents.

On peut, par exemple, disposer d'une classe (éventuellement dérivée d'autres classes) qui résout partiellement un problème donné. On adapte alors cette classe en créant une classe dérivée pour résoudre le problème posé. Dans ce cas, on gagne du temps de programmation puisqu'on réutilise une partie de logiciel. Même si l'on n'exploite pas toutes les fonctions de la classe de départ, on ne sera pas trop pénalisé dans la mesure où les fonctions non utilisées ne seront pas incorporées à l'édition de liens. Le seul risque encouru sera celui d'une perte de temps dans des appels imbriqués que l'on aurait pu limiter en réécrivant totalement la classe. En revanche, les membres donnée non utilisés (s'il y en a) occuperont de l'espace dans tous les objets du type.

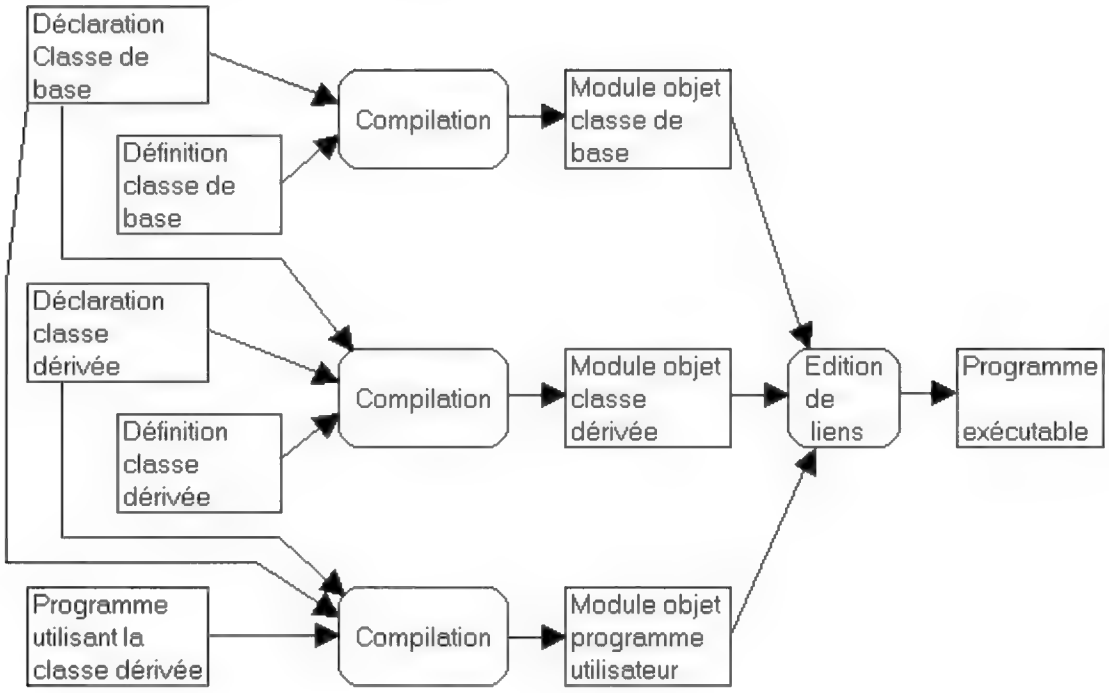
Dans cette catégorie, on peut également ranger le cas où, disposant d'une classe, on souhaite en modifier l'interface utilisateur pour qu'elle réponde à des critères donnés. La classe dérivée fait la même chose que la classe de base ; seule la façon de l'utiliser est différente.

Dans un tout autre esprit, on peut, au contraire, en "partant de rien", chercher à résoudre un problème en l'exprimant sous forme d'un graphe (ou d'un arbre si l'on ne dispose pas de l'héritage multiple) de classes. On peut même être amené à créer ce que l'on nomme des "classes abstraites", c'est-à-dire des classes dont la vocation n'est pas de donner naissance à des objets, mais simplement d'être utilisées comme classes de base pour d'autres classes dérivées.

En ce qui concerne l'utilisation (compilation, édition de liens) d'une classe dérivée au sein d'un programme, les choses sont très simples si la classe de base et la classe dérivée sont créées dans le programme lui-même

(un seul fichier source, un module objet...). Il en ira rarement ainsi. Voici un schéma illustrant la manière de compiler successivement :

- une classe de base,
- une classe dérivée,
- un programme utilisant cette classe dérivée.



Exploitation d'une classe dérivée

En général, les "déclarations" figureront dans un fichier en-tête, introduit par #include pour la compilation.

Naturellement, ce schéma devra être adapté pour les cas où l'on a plus de deux classes ou lorsque la classe de base et la classe dérivée figurent dans un même fichier source (le fichier en-tête correspondant étant alors probablement unique).

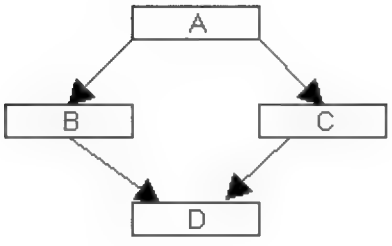
XIV. L'HÉRITAGE MULTIPLE

Comme nous l'avons signalé dans le chapitre précédent, C++ dispose (depuis la version 2) de possibilités d'héritage multiple. Il s'agit là d'une généralisation conséquente, dans la mesure où elle permet de s'affranchir de la contrainte hiérarchique imposée par l'héritage simple.

Malgré tout, son usage reste assez peu répandu ; la principale raison réside certainement dans les difficulés qu'il implique au niveau de la conception des logiciels. Il est, en effet, manifestement plus facile de structurer un ensemble de classes suivant un ou plusieurs "arbres" (cas de l'héritage simple) que suivant un simple "graphe orienté sans circuit" (cas de l'héritage multiple).

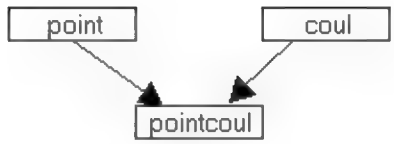
Bien entendu, la plupart des choses que nous avons dites à propos de l'héritage simple se généralisent au cas de l'héritage multiple. Néanmoins, un certain nombre d'informations supplémentaires doivent être introduites pour répondre aux questions suivantes :

- Comment exprimer cette dépendance "multiple", au sein d'une classe dérivée ?
- Comment seront appelés les constructeurs et destructeurs concernés : ordre, transmission d'information, etc. ?
- Comment régler les conflits qui risquent d'apparaître dans des situations telles que celle-ci où D hérite de B et C qui héritent toutes deux de A ?



1. MISE EN ŒUVRE DE L'HÉRITAGE MULTIPLE

Considérons une situation simple, celle où une classe, que nous nommerons pointcoul, hérite de deux autres classes nommées point et coul.



Supposons, pour fixer les idées, que les classes point et coul se présentent ainsi (nous les avons réduites à ce qui était indispensable à la démonstration) :

```
class point
{     int x, y ;
  public :
      point (...) {...}
      ~point () {...}
      affiche () {...}
} ;
```

```
class coul
{     short couleur ;
  public :
      coul (...) {...}
          ~coul () {...}
      affiche () {...}
} ;
```

Nous pouvons définir une classe pointcoul héritant de ces deux classes en la déclarant ainsi (ici, nous avons choisi public pour chacune de nos deux classes, mais, le cas échéant, nous pourrions employer private[1] ou protected[2]).

```
class pointcoul : public point, public coul
{  ...  } ;
```

Notez que nous nous sommes contenté de remplacer la mention d'une classe de base par une liste de mentions de classes de base.

Au sein de cette classe, nous pouvons définir de nouveaux membres. Ici, nous nous limitons à un constructeur, un destructeur et une fonction d'affichage.

Dans le cas de l'héritage simple, le constructeur devait pouvoir retransmettre des informations au constructeur de la classe de base. Il en va de même ici, avec cette différence qu'il y a deux classes de base. L'en-tête du constructeur se présentera ainsi :

```
pointcoul ( .......)  :  point (.......),   coul (.......)
              |                  |                |
          arguments          arguments        arguments
        de pointcoul       à transmettre    à transmettre
                              à point          à coul
```

L'ordre d'appel des constructeurs est le suivant :

- constructeurs des classes de base, dans l'ordre où les classes de base sont déclarées dans la classe dérivée (ici, point puis coul),
- constructeur de la classe dérivée (ici, pointcoul).

---

1 - Depuis la version 2.0.

2 - Depuis la version 3.

Les destructeurs éventuels seront, là encore, appelés dans l'ordre inverse lors de la destruction d'un objet de type pointcoul.

Dans la fonction d'affichage que nous nommerons, elle aussi, affiche, nous vous proposons de lui faire employer successivement les fonctions affiche de point et de coul. Comme dans le cas de l'héritage simple, on peut, dans une fonction membre de la classe dérivée, utiliser toute fonction membre publique (ou protégée) d'une classe de base. Lorsque plusieurs fonctions membre portent le même nom dans différentes classes, on peut lever l'ambiguïté en employant l'opérateur de résolution de portée. Ainsi, ici, notre fonction affiche de pointcoul sera :

```
void affiche ()
{    point::affiche () ; coul::affiche () ;
}
```

Bien entendu, si les fonctions d'affichage de point et de coul se nommaient, par exemple, affp et affc, notre fonction affiche aurait pu s'écrire simplement :

```
void affiche ()
{  affp () ;  affc () ;
}
```

L'utilisation de la classe pointcoul est "classique". Un objet de type pointcoul peut faire appel aux fonctions membre de pointcoul ou, éventuellement, aux fonctions membre des classes de base point et coul (en se servant éventuellement de l'opérateur de résolution de portée pour lever des ambiguïtés). Par exemple, avec :

```
pointcoul p(3, 9, 2) ;
```

p.affiche () appellera la fonction affiche de pointcoul, tandis que p.point::affiche () appellera la fonction affiche de point.

Naturellement, si l'une des classes point et coul était elle-même dérivée d'une autre classe, il serait également possible d'en utiliser l'un des membres (en ayant éventuellement plusieurs fois recours à l'opérateur de résolution de portée).

Voici un exemple complet de définition et d'utilisation de notre classe pointcoul, dans laquelle ont été introduits quelques affichages informatifs.

```
#include <iostream.h>
class point
{  int x, y ;
  public :
   point (int abs, int ord)
     { cout << "++ Constr. point \n" ; x=abs ; y=ord ;
     }
   ~point () { cout << "-- Destr. point \n" ; }
   void affiche ()
    { cout << "Coordonnées : " << x << " " << y << "\n" ;
    }
} ;
class coul
{  short couleur ;
  public :
```

```
   coul (int cl)
     { cout << "++ Constr. coul \n" ; couleur = cl ;
     }
   ~coul () { cout << "-- Destr.  coul \n" ; }
   void affiche ()
     { cout << "Couleur : " << couleur << "\n" ;
     }
} ;
class pointcoul : public point, public coul
{  public :
    pointcoul (int, int, int) ;
    ~pointcoul () { cout << "---- Destr. pointcoul \n" ; }
    void affiche ()
     { point::affiche () ; coul::affiche () ;
     }
} ;
pointcoul::pointcoul (int abs, int ord, int cl) : point (abs, ord), coul (cl)
{  cout << "++++ Constr. pointcoul \n" ;
}

main()
{  pointcoul p(3,9,2) ;
   cout << "------------\n" ;
   p.affiche () ;                  // appel de affiche de pointcoul
   cout << "------------\n" ;
   p.point::affiche () ;           // on force l'appel de affiche de point
   cout << "------------\n" ;
   p.coul::affiche () ;            // on force l'appel de affiche de coul
   cout << "------------\n" ;
}
```

```
++ Constr. point
++ Constr. coul
++++ Constr. pointcoul
------------
Coordonnées : 3 9
Couleur : 2
------------
Coordonnées : 3 9
------------
Couleur : 2
------------
---- Destr. pointcoul
-- Destr.  coul
-- Destr. point
```

Un exemple d'héritage multiple : pointcoul hérite de point et de coul

Remarque :

Nous avons vu comment distinguer deux fonctions membre de même nom appartenant à deux classes différentes (par exemple affiche). La même démarche s'appliquerait à des membres donnée (dans la mesure où leur accès est autorisé). Par exemple, avec :

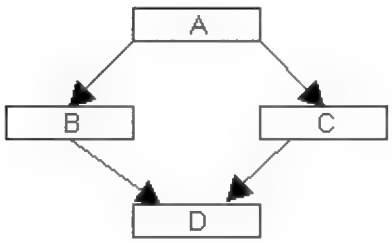
```
class A                                    class B
{     .....                                      {     .....
  public :                                   public :
      int x ;                                          int x ;
      .....                                            .....
} ;                                        } ;

class C : public A, public B
{     .....
} ;
```

C possédera deux membres nommés x, l'un hérité de A, l'autre de B. Au sein des fonctions membre de C, on fera la distinction à l'aide de l'opérateur de résolution de portée : on parlera de A::x ou de B::x.

2. POUR RÉGLER LES ÉVENTUELS CONFLITS : LES CLASSES VIRTUELLES

Considérons la situation ci-après :



correspondant à des déclarations telles que :

```
class A
{     .....
      int x, y ;
} ;
class B : public A {.....} ;
class C : public A {.....} ;
class D : public B, public C
{     .....
} ;
```

On peut dire, en quelque sorte, que D hérite deux fois de A ! Dans ces conditions, les membres de A (fonctions ou données) vont apparaître **deux fois** dans D. En ce qui concerne les fonctions membre, cela est manifestement inutile (ce sont les mêmes fonctions) mais sans importance puisque les fonctions ne sont pas réellement dupliquées (il n'en existe qu'une pour la classe de base). Par contre, en ce qui concerne les

membres donnée (tels que x et y dans notre exemple), ils seront effectivement **dupliqués dans tous les objets de type D**.

Y a-t-il redondance ? En fait, la réponse dépend du problème. Si l'on souhaite que D dispose de deux jeux de données (de A), on ne fera rien de particulier et on se contentera de les distinguer à l'aide de l'opérateur de résolution de portée ; par exemple, ici, on distinguera :

A::B::x de A::C::x

ou, éventuellement, si B et C ne possèdent pas de membre x :

B::x de C::x

En général, cependant, on ne souhaitera pas cette duplication des données. Dans ces conditions, on peut toujours "se débrouiller" pour travailler avec l'un des deux jeux (toujours le même !) mais cela risque d'être fastidieux et dangereux. En fait, vous pouvez demander à C++ de n'incorporer qu'une seule fois les membres de A dans la classe D. Pour cela, il vous faut préciser, dans les déclarations des classes B et C (attention, pas dans celle de D !) que la classe A est "virtuelle" (mot clé virtual) :

```
class B : public virtual A {.....} ;
class C : public virtual A {.....} ;
class D : public B, public C {.....} ;
```

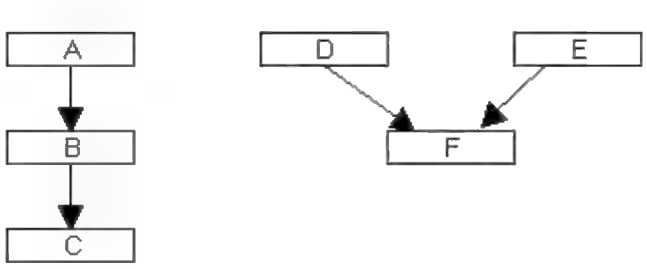
Notez bien que virtual apparaît ici dans B et C. En effet, mentionner A comme "virtuelle" dans la déclaration de B signifie que A ne devra être introduite qu'une seule fois dans les descendants éventuels de C. Autrement dit, cette déclaration n'a guère d'effet sur les classes B et C elles-mêmes (si ce n'est une information "cachée" mise en place par le compilateur pour marquer au sein de B et C, A comme étant virtuelle !). Avec ou sans le mot virtual, les classes B et C, tant qu'elles n'ont pas de descendants, se comportent de la même manière.

Remarque :

Le mot virtual peut être placé indifféremment avant ou après le mot public (ou le mot private).

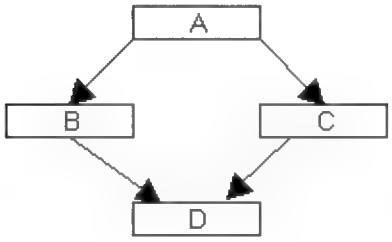
3. APPELS DES CONSTRUCTEURS ET DES DESTRUCTEURS - CAS DES CLASSES VIRTUELLES

Nous avons vu comment sont appelés les constructeurs et les destructeurs dans des situations telles que :

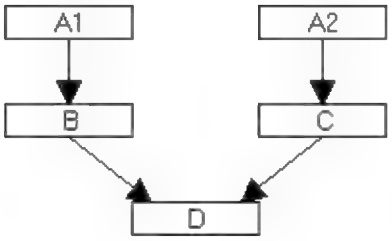


De plus, nous avons vu comment demander des transferts d'information entre un constructeur d'une classe et les constructeurs de ses ascendants directs (C pour B, B pour A, F pour D et E). En revanche, nous ne pouvons pas demander à un constructeur de transférer des informations à un constructeur d'un ascendant

indirect (C pour A, par exemple) et nous n'avons d'ailleurs aucune raison de le vouloir (puisque chaque transfert d'information d'un niveau vers le niveau supérieur était spécifié dans l'en-tête du constructeur du niveau correspondant). Mais considérons maintenant la situation suivante :



Si A n'est pas déclarée virtuelle dans B et C, on peut considérer que, la classe A étant dupliquée, tout se passe comme si l'on était en présence de la situation suivante dans laquelle les notations A_1 et A_2 symbolisent toutes les deux la classe A.



Si D a déclaré les classes B et C dans cet ordre, les constructeurs seront appelés dans l'ordre suivant :

A_1 B A_2 C D

En ce qui concerne les transferts d'information, on peut très bien imaginer que B et C n'aient pas prévu les mêmes choses en ce qui concerne A.

Par exemple, on peut avoir :

```
B (int n, int p, double z) : A (n, p)
C (int q, float x) : A (q)
```

Ceci n'a aucune importance puisque, en définitive, il y aura construction de deux objets distincts de type A.

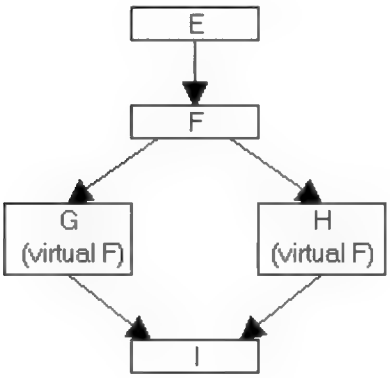
Mais si A a été déclarée virtuelle dans B et C, les choses sont totalement différentes (le dernier schéma n'est plus valable). En effet, dans ce cas, on ne construira qu'un seul objet de type A. Quels arguments faut-il transmettre alors au constructeur ? Ceux prévus par B ou ceux prévus par C ? En fait, C++ résout cette ambiguïté de la façon suivante :

Le choix des informations à fournir au constructeur de A se fait, non plus dans B ou C, mais dans D. Pour ce faire, C++ vous autorise (uniquement dans ce cas) à spécifier, dans le constructeur de D, des informations destinées à A. Ainsi, nous pourrons avoir :

```
D (int n, int p, double z) : B (n, p, z), A (n, p)
```

Bien entendu, il sera inutile de préciser des informations pour A au niveau des constructeurs B et C (comme nous l'avions prévu précédemment, alors que A n'avait pas été déclarée virtuelle dans B et C).

En ce qui concerne l'ordre des appels, **le constructeur d'une classe virtuelle est toujours appelé avant les autres**. Dans notre cas, cela nous conduit à l'ordre (probablement attendu) A, B, C et D. Mais, dans une situation telle que :



cela conduit à l'ordre (moins évident) F, E, G, H, I (ou F, E, H, G, I suivant l'ordre dans lequel figurent G et H dans la déclaration de I).

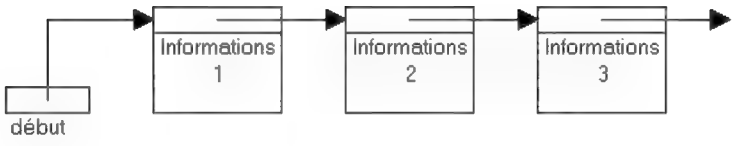
4. EXEMPLE D'UTILISATION DE L'HÉRITAGE MULTIPLE : LISTE CHAÎNÉE DE POINTS

Nous vous proposons ici de réaliser une classe permettant de gérer une liste chaînée d'objets de type point. Certes, nous pourrions en faire une classe à part entière. Mais nous pouvons tout naturellement penser qu'elle pourrait hériter de la classe point. Une réflexion supplémentaire nous conduit alors à penser que certaines choses développées pour une liste chaînée de points doivent pouvoir s'appliquer à une liste chaînée d'autres types d'objets[3]. D'où l'idée de concevoir tout d'abord une classe particulière prenant en charge tout ce qui concerne la gestion de la liste chaînée, sans entrer dans les détails spécifiques aux types des objets concernés. Bien entendu, une telle classe n'aura aucun intérêt en soi ; elle sera uniquement conçue en vue d'être dérivée ; ce sera une classe abstraite.

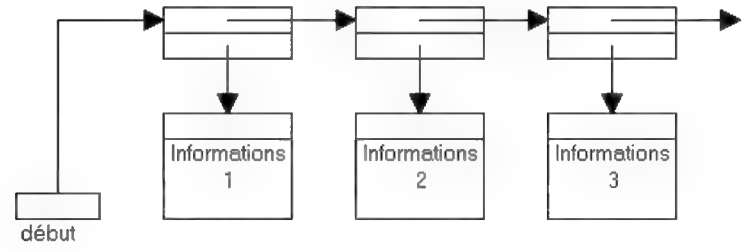
4.1 Une classe abstraite : liste chaînée

Nous nous limiterons ici à une liste chaînée simple, c'est-à-dire définie par un premier élément et où chaque élément comporte un pointeur sur le suivant. Le schéma le plus classique d'une telle liste chaînée est celui-ci :

3 - Mieux, nous pourrions envisager des listes chaînées d'objets quelconques. Dans ce cas, il nous faudra connaître le mécanisme des procédures virtuelles, du moins si nous souhaitons, comme cela est probable, pouvoir "identifier" convenablement chaque objet. Ce mécanisme sera exposé dans le prochain chapitre.



Toutefois, ici, la nature de l'information associée à chaque élément de la liste n'est pas connue. Aussi, faut-il se tourner vers un autre schéma. Celui-ci convient (il porte souvent le nom de liste de "conteneurs[4]") :



Bien entendu, notre classe liste se contentera de gérer des éléments simples réduits chacun à :

- un pointeur sur l'élément suivant,
- un pointeur sur l'information associée (en fait, ici, un objet).

On voit donc que notre classe va posséder, au moins :

- un membre donnée : pointeur sur le premier élément (debut, dans notre schéma[5]),
- une fonction membre destinée à insérer dans la liste (nous choisirons en début de liste, par souci de simplification) un objet dont on lui fournira l'adresse. Notez que, au sein de la classe liste, cette adresse doit être de type void * (puisque l'on souhaite pouvoir gérer n'importe quel type d'objet).

D'où une première ébauche de notre classe liste :

```
struct element                          // structure d'un élément de liste
{ element * suivant ;                   // pointeur sur l'élément suivant
  void * contenu ;                      // pointeur sur un objet quelconque
} ;
class liste
{ element * debut ;                     // pointeur sur premier élément

 public :
  liste () ;                            // constructeur
  ~liste () ;                           // destructeur
  void ajoute (void *) ;                // ajoute un élément en début de liste
   .....
} ;
```

Bien entendu, nous allons avoir besoin d'autres possibilités, telles que :

- afficher les objets pointés par la liste,

4 - De l'anglais "containers list".

5 - Attention, ici, les objets de notre classe liste seront des listes et non des éléments de liste !

- rechercher un élément,
- supprimer un élément,
- etc.

Ces actions pourraient effectivement être réalisées par la classe liste elle-même, si elle connaissait la nature des objets. Ce n'est toutefois pas le cas avec la démarche que nous avons adoptée.

Cependant, les activités évoquées font toutes appel à un mécanisme de "parcours" de la liste. Certes, ce parcours devra pouvoir être contrôlé (initialisé, poursuivi, interrompu...) depuis "l'extérieur" de la liste ; mais il est possible de prévoir des fonctions élémentaires telles que :

- initialiser le parcours,
- avancer d'un élément.

Celles-ci nécessitent un "pointeur sur un élément courant" ; il sera membre donnée de notre classe liste ; nous le nommerons courant. Par ailleurs, les deux fonctions membre évoquées peuvent fournir en retour une information concernant l'objet courant : à ce niveau, on peut choisir entre :

- l'adresse de l'élément courant,
- l'adresse de l'objet courant,
- la valeur de l'élément courant.

La première solution contient la seconde ; si ptr est l'adresse de l'élément courant, ptr-> contenu fournit celle de l'objet courant. Malgré tout, on pourrait objecter que l'utilisateur de la classe n'a pas à accéder aux éléments de la liste.

La troisième solution peut paraître plus sûre. En fait, elle n'empêche pas l'utilisateur d'aller examiner lui-même d'autres éléments de la liste et de les modifier (il n'a que la "valeur" de l'élément courant, mais celle-ci lui donne accès aux "adresses" de tous les éléments suivants).

Choisissons donc la seconde solution. Dans ce cas, l'utilisateur n'a alors plus de moyen de détecter la fin de liste. Nous prévoirons donc une fonction supplémentaire permettant de savoir si la fin de liste est atteinte (en toute rigueur, nous aurions pu également convenir de fournir dans ce cas, un pointeur nul comme adresse de l'objet courant ; mais ce serait moins pratique car il faudrait obligatoirement agir sur le pointeur de liste avant de savoir si l'on est à la fin).

En définitive, nous introduisons trois nouvelles fonctions membre :

```
void * premier () ;
void * prochain () ;
int fini () ;
```

Voici, ci-après, la liste définitive de notre classe liste

```
#include <stddef.h>                        // pour la définition de NULL
// ********************* classe liste ************************************
struct element                             // structure d'un élément de liste
{ element * suivant ;                      // pointeur sur l'élément suivant
  void * contenu ;                         // pointeur sur un objet quelconque
} ;
class liste
{ element * debut ;                        // pointeur sur premier élément
```

```
  element * courant ;                    // pointeur sur élément courant
 public :
  liste ()                               // constructeur
    { debut = NULL ; courant = debut ; }
  ~liste () ;                            // destructeur
  void ajoute (void *) ;                 // ajoute un élément en début de liste
  void premier ()                        // positionne sur premier élément
    { courant = debut ; }
  void * prochain ()      // fournit l'adresse de l'élément courant (0 si
inexistant)
                          //  et se positionne sur le prochain élément (rien si
fin)
    { void * adel = NULL ;
      if (courant != NULL){ adel = courant -> contenu ;
                            courant = courant -> suivant ;
                          }
      return adel ;
    }
  int fini () { return (courant == NULL) ; }
} ;
liste::~liste ()
{ element * suiv ;
  courant = debut ;
  while (courant != NULL )
   { suiv = courant->suivant ; delete courant ; courant = suiv ; }
}
void liste::ajoute (void * chose)
{ element * adel = new element ;
  adel->suivant = debut ;
  adel->contenu = chose ;
  debut = adel ;
}
```

Une classe abstraite : liste chaînée

Voyez la manière dont nous avons programmé le destructeur ~liste. Celui-ci libère tous les emplacements alloués pour les éléments de la classe liste. Il n'a toutefois aucune action sur les objets pointés. Pour qu'il en soit ainsi, il faudrait faire des hypothèses sur la manière dont les objets sont créés. Dans le cas d'objets dynamiques, on pourrait envisager de les faire détruire par ~liste. Encore faudrait-il pouvoir régler deux problèmes :

- connaître le type de ces objets. Ce n'est pas le cas ici. Nous verrons toutefois que ce point peut être résolu par l'utilisation de procédures virtuelles,
- être en mesure de savoir si la destruction d'un pointeur sur un objet nous autorise à détruire l'objet lui-même. Il faut pour cela que l'objet concerné ne soit pas référencé par ailleurs. Une solution générale[6] à ce problème réside, comme nous l'avons déjà dit, dans l'utilisation d'un "compteur de références".

6 - C'est-à-dire ne faisant appel à aucune hypothèse particulière.

Remarque :

Nous aurions pu prévoir une fonction membre chargée de supprimer un élément courant. Cela n'aurait rien apporté de nouveau sur le plan de la conception et aurait donc alourdi inutilement notre exemple. Bien entendu, cette fonction serait indispensable dans une classe "en vraie grandeur".

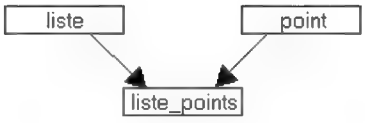
4.2 Création par héritage multiple d'une classe liste_points

Supposons que nous disposions, par ailleurs, d'une classe point que nous limiterons ici au strict minimum :

```
#include <iostream.h>
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  void affiche () { cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;
```

La classe point

Nous allons maintenant chercher à créer une nouvelle classe nous permettant de gérer une liste chaînée d'objets de type point. Il vient alors (presque) naturellement l'idée de créer une classe liste_points héritant simultanément de liste et de point :



A ce propos, il est intéressant de noter que cet héritage apparemment naturel conduit à introduire dans la classe liste_points deux membres donnée (x et y) n'ayant pas vraiment d'intérêt : les objets concernés seront créés de manière indépendante de l'objet de type liste_points (on pourrait dire qu'on créera ainsi un point pour rien !).

Supposons que nous souhaitions, dans la classe liste_points, pouvoir simplement :

- introduire un nouveau point en début de liste,
- afficher tous les objets de la liste.

Manifestement, la première fonction est automatiquement assurée par la fonction membre ajoute de la classe liste. Il n'est même pas nécessaire de la surdéfinir.

Quant à la fonction d'affichage que nous nommerons affiche, elle va exploiter :

- les fonctions premier, prochain et fini pour explorer toute la liste,
- la fonction affiche de la classe point pour afficher le contenu d'un point.

Voici notre classe liste_points :

```
class liste_points : public liste, public point
{ public :
   liste_points () {}
   void affiche () ;
} ;
void liste_points::affiche ()
{ point * ptr = (point *) premier() ;
  while ( ! fini() ) {  ptr->affiche () ; ptr = (point *) prochain() ; }
}
```

La classe liste_points, héritant de liste et de point

A simple titre de commodité de lecture de l'ensemble, nous vous fournissons à nouveau, de façon groupée, la liste de nos trois classes accompagnées d'un exemple d'utilisation.

```
#include <iostream.h>
#include <stddef.h>                          // pour la définition de NULL

// ********************* classe liste ************************************
struct element                               // structure d'un élément de liste
{ element * suivant ;                        // pointeur sur l'élément suivant
  void * contenu ;                           // pointeur sur un objet quelconque
} ;

class liste
{ element * debut ;                          // pointeur sur premier élément
  element * courant ;                        // pointeur sur élément courant
 public :
  liste ()                                   // constructeur
    { debut = NULL ; courant = debut ;
    }
  ~liste () ;                                // destructeur
  void ajoute (void *) ;                     // ajoute un élément en début de liste
  void premier ()                            // positionne sur premier élément
    { courant = debut ;
    }
  void * prochain ()      // fournit l'adresse de l'élement courant (0 si
inexistant)
                          //  et se positionne sur le prochain élément (rien si
fin)
    { void * adel = NULL ;
      if (courant != NULL){ adel = courant -> contenu ;
                            courant = courant -> suivant ;
                          }
      return adel ;
    }
  int fini () { return (courant == NULL) ; }
```

```
} ;
liste::~liste ()
{ element * suiv ;
  courant = debut ;
  while (courant != NULL )
   { suiv = courant->suivant ; delete courant ; courant = suiv ; }
}
void liste::ajoute (void * chose)
{ element * adel = new element ;
  adel->suivant = debut ; adel->contenu = chose ;
  debut = adel ;
}

// ***************** classe point ****************************************
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  void affiche () { cout << "Coordonnées : " << x << " " << y << "\n" ; }
} ;

// ***************** classe liste_points *********************************
class liste_points : public liste, public point
{ public :
   liste_points () {}
   void affiche () ;
} ;
void liste_points::affiche ()
{ premier() ;
  while ( ! fini() ) {  point * ptr = (point *) prochain() ; ptr->affiche () ;
}
}


// ***************** programme d'essai de liste_points ***********************
main()
{ liste_points l ;
  point a(2,3), b(5,9), c(0,8) ;
  l.ajoute (&a) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&b) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&c) ; l.affiche () ; cout << "---------\n" ;
}
```

```
Coordonnées : 2 3
---------
Coordonnées : 5 9
Coordonnées : 2 3
---------
Coordonnées : 0 8
Coordonnées : 5 9
Coordonnées : 2 3
---------
```

Exemple d'utilisation de la classe liste_points

XV. LES FONCTIONS VIRTUELLES ET LE TYPAGE DYNAMIQUE

Nous avons vu qu'en C++ un pointeur sur un type d'objet pouvait recevoir l'adresse de n'importe quel objet descendant. Toutefois, cet avantage se trouvait compensé par une lacune importante : l'appel d'une méthode pour un objet pointé conduisait systématiquement à appeler la méthode correspondant au type du pointeur et non pas au type effectif de l'objet pointé lui-même (revoyez éventuellement le paragraphe 6.3 du chapitre XIII).

Cette lacune provient essentiellement de ce que, dans les situations rencontrées jusqu'ici, C++ réalise ce que l'on nomme une ligature statique[1], ou encore un typage statique. Le type d'un objet (pointé) y est déterminé au moment de la compilation. Dans ces conditions, le mieux que puisse faire le compilateur est effectivement de considérer que l'objet pointé a le type du pointeur.

Pour pouvoir obtenir l'appel de la méthode correspondant au type de l'objet pointé, il est nécessaire que le type de l'objet ne soit pris en compte qu'au moment de l'exécution (le type de l'objet désigné par un même pointeur pourra varier au fil du déroulement du programme). On parle alors de ligature dynamique[2] ou de typage dynamique.

Comme nous allons le voir maintenant, en C++, le typage dynamique peut être mis en œuvre en faisant appel au mécanisme des fonctions virtuelles.

1. RAPPEL D'UNE SITUATION OÙ LE TYPAGE DYNAMIQUE EST NÉCESSAIRE

Considérons la situation suivante, déjà rencontrée dans le chapitre XIII :

```
class point
{     .....
      void affiche () ;
      .....
```

```
class pointcol : public point
{     .....
void affiche () ;
      .....
```

1 - En anglais, "early binding".

2 - En anglais, "late binding" ou encore "dynamic binding".

```
} ;                                    } ;

point p ;
pointcol pc ;
point * adp = &p ;
```

L'instruction :

```
adp -> affiche () ;
```

appelle alors la méthode affiche du type point.

Mais si nous exécutons cette affectation (autorisée) :

```
adp = & pc ;
```

le pointeur adp pointe maintenant sur un objet de type pointcol. **Néanmoins**, l'**instruction :**

```
adp -> affiche () ;
```

fait toujours appel à la méthode affiche du type point alors que le type pointcol dispose, lui aussi, d'une méthode affiche.

En effet, comme nous l'avons évoqué en introduction, le choix de la méthode à appeler a été réalisé lors de la compilation ; il a donc été fait en fonction du type de la variable adp. C'est la raison pour laquelle on parle de "ligature statique".

2. LE MÉCANISME DES FONCTIONS VIRTUELLES

Le mécanisme des fonctions virtuelles proposé par C++ va nous permettre de faire en sorte que l'instruction :

```
adp -> affiche ()
```

appelle, non plus systématiquement la méthode affiche de point, mais celle correspondant au type de l'objet réellement désigné par adp (ici point ou pointcol).

Pour ce faire, il suffit tout simplement de déclarer "virtuelle" (mot clé virtual) la méthode affiche de la classe point :

```
class point
{     .....
      virtual void affiche () ;
      .....
} ;
```

Ceci précise au compilateur que les éventuels appels de la fonction affiche doivent utiliser une ligature dynamique et non plus une ligature statique. Autrement dit, lorsque le compilateur rencontrera un appel tel que :

```
adp -> affiche () ;
```

il ne décidera pas de la procédure à appeler. Il se contentera de mettre en place un dispositif permettant de n'effectuer le choix de la fonction qu'au moment de l'exécution de cette instruction[3], ce choix étant basé sur le type exact de l'objet ayant effectué l'appel.

Dans la classe pointcol, on ne procédera à aucune modification : il n'est pas nécessaire de déclarer virtuelle dans les classes dérivées une fonction déclarée virtuelle dans une classe de base (cette information serait en fait redondante).

A titre d'exemple, le programme suivant correspond à celui du paragraphe 6.3 du chapitre XIII, dans lequel nous nous sommes contenté de rendre virtuelle la fonction affiche.

```
#include <iostream.h>
class point
{ protected :            // pour que x et y soient accessibles à pointcol
   int x, y ;
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   virtual void affiche ()
     { cout << "Je suis un point \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y << "\n" ;
     }
} ;
class pointcol : public point
{  short couleur ;
  public :
   pointcol (int abs=0, int ord=0, short cl=1) : (abs, ord)
     { couleur = cl ;
     }
   void affiche ()
     { cout << "Je suis un point coloré \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y ;
       cout << "   et ma couleur est :    " << couleur << "\n" ;
     }
} ;

main()
{  point p(3,5) ; point * adp = &p ;
   pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
   adp->affiche () ; adpc->affiche () ;
   cout << "-----------------\n" ;
   adp = adpc ;              //  adpc = adp serait rejeté
   adp->affiche () ; adpc->affiche () ;
}
```

```
Je suis un point
   mes coordonnées sont : 3 5
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
-----------------
```

3 - Plusieurs exécutions de cette même instruction pouvant appeler des fonctions différentes.

```
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
```

Mise en œuvre de ligature dynamique (ici pour affiche) par la technique des fonctions virtuelles

Remarques :

1) Par défaut, C++ met en place des ligatures statiques. A l'aide du mot virtual, on peut choisir la ou les fonctions pour lesquelles on souhaite mettre en place une ligature dynamique.

2) Cet aspect ligature dynamique est limité à un ensemble de classes dérivées les unes des autres.

3. UNE AUTRE SITUATION OÙ LA LIGATURE DYNAMIQUE EST INDISPENSABLE

Dans notre précédent exemple, nous pouvons dire que, lors de la conception de la classe point, nous avons prévu que chacune de ses descendantes redéfinirait à sa guise la fonction affiche. Ceci conduit à prévoir, dans chaque fonction, des instructions d'affichage des coordonnées. Pour éviter cette redondance[4], nous pouvons définir notre fonction affiche (de la classe point) de manière qu'elle :

- affiche les coordonnées (action commune à toutes les classes),
- fasse appel à une autre fonction (nommée, par exemple, identifie), ayant pour vocation d'afficher les informations spécifiques à chaque objet. Bien entendu, ce faisant, nous supposons que chaque descendante de point redéfinira identifie de façon appropriée (mais elle n'aura plus à prendre en charge l'affichage des coordonnées).

Cette démarche nous conduit à définir notre classe point de la façon suivante :

```
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }

  void identifie ()
    { cout << "Je suis un point \n" ; }
  void affiche ()
    { identifie () ;
      cout << "Mes coordonnées sont : " << x << " " << y << "\n" ;
    }
} ;
```

Dérivons une classe pointcol en redéfinissant comme voulu la fonction identifie :

```
class pointcol : public point
{ short couleur ;
 public :
```

4 - Bien entendu, ici, l'enjeu est très limité. Mais il pourrait être important dans un cas réel.

```
  pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs, ord)
    { couleur = cl ; }
  void identifie ()
    { cout << "Je suis un point coloré de couleur : " << couleur << "\n" ; }
} ;
```

Si nous cherchons alors à utiliser pointcol de la façon suivante :

```
pointcol pc (8, 6, 2) ;
pc.affiche () ;
```

nous obtenons alors le résultat :

```
Je suis un point
Mes coordonnées sont : 8 6
```

ce qui n'est manifestement pas ce que nous espérions !

Certes, la compilation de l'appel :

```
pc.affiche ()
```

a conduit le compilateur à appeler la fonction affiche de la classe point (puisque cette fonction n'est pas redéfinie dans pointcol). En revanche, à ce moment-là, l'appel :

```
identifie ()
```

figurant dans cette fonction a *déjà été compilé* en un appel... d'identifie de la classe point.

Comme vous le constatez, bien qu'ici la fonction affiche ait été appelée explicitement pour un objet (et non, comme précédemment, à l'aide d'un pointeur), nous nous trouvons à nouveau en présence d'un problème de ligature statique.

Pour le résoudre, il nous suffit, dans la classe point, de déclarer virtuelle la fonction identifie. Cela permettra au compilateur de mettre en place les instructions assurant l'appel de la fonction identifie correspondant au type de l'objet l'ayant effectivement appelée. Ici, vous noterez cependant que la situation est légèrement différente de celle qui nous a servi à présenter les fonctions virtuelles (paragraphe 1). En effet, l'appel d'identifie est réalisé, cette fois, non plus directement par l'objet lui-même, mais indirectement par la fonction affiche. Nous verrons comment le mécanisme des fonctions virtuelles est également capable de prendre en charge cet aspect.

Voici un programme complet reprenant les définitions de nos classes point et pointcol. Nous y montrons comment un appel tel que pc.affiche () entraîne bien l'appel de identifie du type pointcol (ce qui constituait le but de ce paragraphe). A titre indicatif, nous y avons également introduit quelques appels par pointeur, afin de montrer que, là aussi, les choses se déroulent convenablement.

```
#include <iostream.h>
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  virtual void identifie ()
    { cout << "Je suis un point \n" ; }
```

```
    void affiche ()
      { identifie () ;
        cout << "Mes coordonnées sont : " << x << " " << y << "\n" ;
      }
} ;
class pointcol : public point
{ short couleur ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs, ord)
    { couleur = cl ; }
  void identifie ()
    { cout << "Je suis un point coloré de couleur : " << couleur << "\n" ; }
} ;

main()
{ point p(3,4) ; pointcol pc(5,9,5) ;
  p.affiche () ; pc.affiche () ;      cout << "---------------\n" ;
  point * adp = &p ; pointcol * adpc = &pc ;
  adp->affiche () ; adpc->affiche () ; cout << "---------------\n" ;
  adp = adpc ;
  adp->affiche () ; adpc->affiche () ;
}
```

```
Je suis un point
Mes coordonnées sont : 3 4
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
---------------
Je suis un point
Mes coordonnées sont : 3 4
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
---------------
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
```

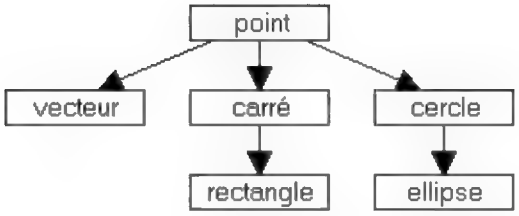
Mise en œuvre de ligature dynamique (ici pour identifie) par la technique des fonctions virtuelles

4. LES FONCTIONS VIRTUELLES EN GÉNÉRAL

Les deux précédents exemples constituaient des cas particuliers d'utilisation de méthodes virtuelles. Nous vous proposons ici de voir quelles en sont les possibilités et les limitations.

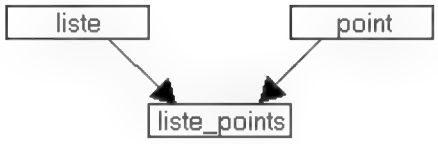
4.1 Leurs limitations sont celles de l'héritage

A partir du moment où une fonction f a été déclarée virtuelle dans une classe A, elle sera soumise à la ligature dynamique dans A et dans toutes les classes descendantes de A : on n'est donc pas limité aux descendantes directes. Ainsi, on peut imaginer une hiérarchie[5] de formes géométriques :



Si la fonction affiche est déclarée virtuelle dans la classe point et redéfinie dans les autres classes descendant de point, elle sera bien soumise à la ligature dynamique. Il est même envisageable que les six classes ci-dessus soient parfaitement définies et compilées et qu'on vienne en ajouter de nouvelles, sans remettre en cause les précédentes de quelque façon que ce soit. Ce dernier point serait d'ailleurs encore plus flagrant si, comme dans notre second exemple (paragraphe 3), la fonction affiche, non virtuelle, faisait elle-même appel à une fonction virtuelle identifie, redéfinie dans chaque classe. En effet, dans ce cas, on voit que la fonction affiche aurait pu être réalisée et compilée (au sein de point), sans que toutes les fonctions identifie qu'elle était susceptible d'appeler soient connues. On trouve là un aspect séduisant de réutilisabilité : on a défini dans affiche un certain "scénario" dont certaines parties pourront être explicitées plus tard, lors de la création de classes dérivées.

De même, si l'on dispose de l'héritage multiple et que l'on a défini cette structure (comme dans le paragraphe 4 du chapitre XIV) :



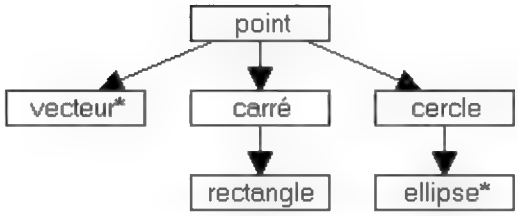
Si, dans point, la fonction affiche a été déclarée virtuelle, il devient possible d'utiliser la classe liste-points pour gérer une liste d'objets "hétérogènes" en dérivant les classes voulues de point et en y redéfinissant affiche. Nous en verrons un exemple dans le paragraphe suivant.

4.2 La redéfinition d'une fonction virtuelle n'est pas obligatoire

Jusqu'ici, nous avons toujours redéfini dans les classes descendantes une méthode déclarée virtuelle dans une classe de base. Cela n'est pas plus indispensable que dans le cas des fonctions membre ordinaires. Ainsi, considérons à nouveau la précédente hiérarchie de figures, en supposant que affiche n'a été redéfinie que dans les classes que nous avons marquées d'une étoile (et définie, bien sûr, comme virtuelle dans point).

---

5 - Mais, depuis la version 2.0, compte tenu des possibilités d'héritage multiple, on peut envisager n'importe quel graphe orienté.



Dans ces conditions, l'appel d'affiche conduira, pour chaque classe, à l'appel de la fonction mentionnée à côté :

vecteur	vecteur::affiche
carre	point::affiche
rectangle	point::affiche
cercle	point::affiche
ellipse	ellipse::affiche

Le même mécanisme s'appliquerait en cas d'héritage multiple, à condition de le compléter, le cas échéant, par les règles concernant les ambiguïtés.

4.3 Fonctions virtuelles et surdéfinition

On peut surdéfinir[6] une fonction virtuelle, chaque fonction surdéfinie pouvant être ou ne pas être déclarée virtuelle.

Par ailleurs, si l'on a défini une fonction virtuelle dans une classe et qu'on la surdéfinit dans une classe dérivée avec des arguments différents, il s'agira alors bel et bien d'une **autre fonction**. Si cette dernière n'est pas déclarée virtuelle, elle sera, quant à elle, soumise à une ligature statique.

En fait, on peut considérer que le statut virtuel/non virtuel joue, lui aussi, un rôle discriminateur dans le choix d'une fonction surdéfinie. D'une manière générale, par souci de simplicité et de lisibilité, nous ne saurions trop vous conseiller d'éviter d'exploiter cette possibilité. Plus précisément, si vous devez surdéfinir une fonction virtuelle, il est préférable que toutes les fonctions de même nom restent virtuelles.

4.4 On peut déclarer une fonction virtuelle dans n'importe quelle classe

Dans tous nos exemples, nous avions déclaré virtuelle une fonction d'une classe qui n'était pas, elle-même, dérivée d'une autre. Cela n'est pas obligatoire. Ainsi, dans nos exemples de hiérarchie de formes, point pourrait elle-même dériver d'une autre classe. Dans ce cas, cependant, deux situations doivent être distinguées :

6 - Ne confondez pas surdéfinition et redéfinition.

- la fonction affiche de la classe point n'a jamais été définie dans les classes ascendantes : aucun problème particulier ne se pose,
- la fonction affiche a déjà été définie, avec les mêmes arguments, dans une classe ascendante. Dans ce cas, il faut considérer la fonction virtuelle affiche comme une nouvelle fonction (comme s'il y avait eu surdéfinition – le caractère virtuel/non virtuel servant, comme nous l'avons déjà dit, à faire la distinction). Bien entendu, toutes les nouvelles définitions d'affiche dans les classes descendantes seront soumises à la ligature dynamique, sauf si l'on effectue un appel explicite d'une fonction d'une classe ascendante au moyen de l'opérateur de résolution de portée. Rappelons toutefois que nous vous déconseillons fortement ce type de situation.

4.5 Quelques restrictions

Seule une fonction membre peut être virtuelle. Cela se justifie par le mécanisme employé pour effectuer la ligature dynamique, à savoir un choix basé sur le type de l'objet ayant appelé la fonction ; cela ne pourrait manifestement pas s'appliquer à une fonction "ordinaire" (même si elle était amie d'une classe).

Un constructeur ne peut pas être virtuel. En revanche, un destructeur peut l'être.

5. LES FONCTIONS VIRTUELLES PURES : UN OUTIL POUR LA CRÉATION DE CLASSES ABSTRAITES

Nous avons déjà eu l'occasion de dire qu'on pouvait être amené à définir des classes destinées, non pas à instancier des objets, mais simplement à donner naissance à d'autres classes par héritage. En P.O.O., on dit qu'on a affaire à des "classes abstraites".

En C++, vous pouvez toujours définir de telles classes. Mais il se peut que vous soyez amené à y introduire certaines fonctions virtuelles dont vous ne pouvez encore donner aucune définition. Imaginez, par exemple une classe abstraite forme_geo, destinée à gérer le dessin sur un écran de différentes formes géométriques (carré, cercle...). Supposez que vous souhaitiez déjà y faire figurer une fonction deplace destinée à déplacer une figure. Il est probable que celle-ci fera alors appel à une fonction d'affichage de la figure (nommée, par exemple, dessine). La fonction dessine sera déclarée virtuelle dans la classe forme_geo et devra être redéfinie dans ses descendants. Mais quelle définition lui fournir dans forme_geo ? Avec ce que vous connaissez de C++, vous avez toujours la ressource de prévoir une définition vide[7] :

Toutefois, deux lacunes apparaissent alors :

a) Rien n'interdit à un utilisateur de déclarer un objet de classe forme_geo, alors que dans l'esprit du concepteur, il s'agissait d'une classe abstraite. L'appel de deplace pour un tel objet conduira à un appel de dessine ne faisant rien ; même si aucune erreur n'en découle, cela n'a guère de sens !

b) Rien n'oblige une classe descendant de forme_geo à redéfinir dessine. Si elle ne le fait pas, on retrouve les problèmes évoqués ci-dessus.

C++ vous propose (depuis la version 2.0) un outil facilitant la définition de classes abstraites : il s'agit des "fonctions virtuelles pures". Ce sont des fonctions virtuelles dont la définition est `nulle` (0), et non plus seulement vide. Par exemple, nous aurions pu faire de notre fonction dessine de la classe forme_geo une fonction virtuelle pure en la déclarant[8] ainsi :

---

7 - Notez bien qu'il vous faut absolument définir dessine dans forme_geo puisqu'elle est appelée par deplace.

8 - Ici, on ne peut plus distinguer déclaration et définition.

```
virtual void dessine (...) = 0 ;
```

Certes, à ce niveau, l'intérêt de cette convention n'apparaît pas encore. Mais, en fait, C++ adopte les règles suivantes :

- une classe comportant au moins une fonction virtuelle pure est considérée comme abstraite et il n'est plus possible de déclarer des objets de son type,
- une fonction déclarée virtuelle pure dans une classe de base doit obligatoirement être redéfinie[9] dans une classe dérivée ou déclarée à nouveau virtuelle pure[10] ; dans ce dernier cas, la classe dérivée est, elle aussi, abstraite.

Comme vous le voyez, l'emploi de fonctions virtuelles pures règle les deux problèmes soulevés par l'emploi de définitions vides. Dans le cas de notre classe forme_geo, le fait d'avoir rendu dessine virtuelle pure interdit :

- la déclaration d'objets de type forme_geo,
- la définition de classes dérivées de forme_geo, dans lesquelles on omettrait la définition de dessine.

Remarque :

La notion de fonction virtuelle pure dépasse celle de classe abstraite. Si C++ s'était contenté de déclarer une classe comme étant abstraite, ceci n'aurait servi qu'à en interdire l'utilisation ; il aurait fallu une seconde convention pour préciser les fonctions devant obligatoirement être redéfinies.

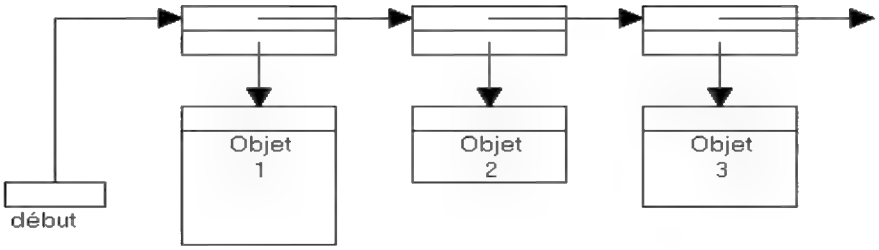
6. EXEMPLE D'UTILISATION DE FONCTIONS VIRTUELLES : LISTE HÉTÉROGÈNE

Dans le programme du paragraphe 4 du chapitre XIV, nous vous avons fourni un exemple de gestion d'une liste chaînée de points. Tous les objets de la liste étaient du même type : on dit qu'on avait affaire à une liste homogène.

Telle que nous avons conçu notre classe liste, nous aurions pu l'employer pour créer (par dérivation) d'autres classes permettant la gestion d'objets d'un autre type. Mais peut-on aller plus loin et définir une classe permettant de gérer une liste comportant des objets de type différent ? Compte tenu de la manière dont notre classe liste a été conçue, cela correspondrait à un schéma de ce type :

9 Toujours avec les mêmes arguments, sinon il s'agit d'une autre fonction.

10 - Depuis la version 3.0, si une fonction virtuelle pure d'une classe de base n'est pas redéfinie dans une classe dérivée, elle reste une fonction virtuelle pure de cette classe dérivée ; dans les versions antérieures, on obtenait une erreur.



Certes, la nature variable du type des objets ne présente pas de difficultés puisqu'ils sont repérés par des pointeurs de type void *. En revanche, on voit que l'affichage de la liste va devoir être en mesure d'appliquer à chaque objet la méthode appropriée. Ceci implique la mise en œuvre de la ligature dynamique : la fonction d'affichage sera définie virtuelle dans un premier type d'objet (ici point) et redéfinie dans chacun de ses descendants.

En définitive, on voit qu'on va pouvoir gérer une liste d'objets de types différents sous réserve que les classes correspondantes soient toutes dérivées d'une même classe de base. Cela peut sembler quelque peu restrictif. En fait, cet aspect "famille de classes" peut toujours être obtenu par la création d'une classe abstraite (réduite au minimum, éventuellement à une fonction affiche vide ou virtuelle pure) destinée simplement à donner naissance aux classes concernées ; bien entendu, ceci n'est concevable que si les classes en question ne sont pas déjà figées (car il faut qu'elles héritent de cette classe abstraite).

Nous vous proposons, à titre d'exemple, une autre utilisation de la classe liste du paragraphe 4 du chapitre XIV. Nous lui avons adjoint une classe abstraite mere, destinée à donner naissance aux types susceptibles d'être gérés par notre liste. Ici, nous avons employé la possibilité de définir une fonction virtuelle pure (affiche). Nous avons complété notre classe liste avec une fonction d'affichage (affiche_liste) de la liste.

A simple titre indicatif, nous avons défini deux classes point et complexe (lesquelles n'ont pas besoin de dériver l'une de l'autre), dérivées de la classe abstraite mere et dotées chacune d'une fonction affiche appropriée. Voici un exemple complet :

```
#include <iostream.h>
#include <stddef.h>                        // pour la définition de NULL
// ***************** classe mere ***************************************
class mere
{ public :
   virtual void affiche () = 0 ;    // fonction virtuelle pure
} ;

// ********************* classe liste ************************************
struct element                             // structure d'un élément de liste
{ element * suivant ;                      // pointeur sur l'élément suivant
  void * contenu ;                         // pointeur sur un objet quelconque
} ;

class liste
{ element * debut ;                        // pointeur sur premier élément
  element * courant ;                      // pointeur sur élément courant
 public :
```

```
    liste ()                                  // constructeur
      { debut = NULL ; courant = debut ; }
    ~liste () ;                               // destructeur
    void ajoute (void *) ;                    // ajoute un élément
    void premier ()                           // positionne sur premier élément
      { courant = debut ;
      }
    void * prochain ()            // fournit l'adresse de l'élément courant (0 si
  fin)
                                  // et positionne sur prochain élément (rien si
  fin)
      { void * adsuiv = NULL ;
        if (courant != NULL){ adsuiv = courant -> contenu ;
                              courant = courant -> suivant ;
                            }
        return adsuiv ;
      }
    void affiche_liste () ;                   // affiche tous les éléments de la
  liste
    int fini () { return (courant == NULL) ; }
  } ;
  liste::~liste ()
  { element * suiv ;
    courant = debut ;
    while (courant != NULL )
     { suiv = courant->suivant ; delete courant ; courant = suiv ; }
  }
  void liste::ajoute (void * chose)
  { element * adel = new element ;
    adel->suivant = debut ;
    adel->contenu = chose ;
    debut = adel ;
  }


  void liste::affiche_liste ()
  { mere * ptr ;       // attention, mere * et pas void *
    premier() ;
    while ( ! fini() )
      { ptr = (mere *) prochain() ;
        ptr->affiche () ;
      }
  }
  // ***************** classe point ***************************************
  class point : public mere
  { int x, y ;
   public :
    point (int abs=0, int ord=0) { x=abs ; y=ord ; }
    void affiche ()
      { cout << "Point de coordonnées : " << x << " " << y << "\n" ; }
  } ;
  // ***************** classe complexe ************************************
  class complexe : public mere
  { double reel, imag ;
   public :
```

```
  complexe (double r=0, double i=0) { reel=r ; imag=i ; }
  void affiche ()
    { cout << "Complexe : " << reel << " + " << imag << "i\n" ; }
} ;
// ***************** programme d'essai ************************************
main()
{ liste l ;
  point a(2,3), b(5,9) ;
  complexe x(4.5,2.7), y(2.35,4.86) ;
  l.ajoute (&a) ; l.ajoute (&x) ; l.affiche_liste () ; cout << "--------\n" ;
  l.ajoute (&y) ; l.ajoute (&b) ; l.affiche_liste () ;
}
Complexe : 4.5 + 2.7i
Point de coordonnées : 2 3
--------
Point de coordonnées : 5 9
Complexe : 2.35 + 4.86i
Complexe : 4.5 + 2.7i
Point de coordonnées : 2 3
```

Utilisation de fonctions virtuelles : liste hétérogène

Remarque :

Dans la classe liste, beaucoup de pointeurs de type void * pourraient être remplacés par des pointeurs de type mere * (pour peu qu'on ne cherche pas à utiliser cette même classe pour des objets d'un type non dérivé de mere). En revanche, le pointeur ptr utilisé dans affiche_liste doit bien rester de type mere * car c'est sur lui que repose ici le mécanisme de la ligature dynamique de la fonction affiche.

7. LE MÉCANISME D'IDENTIFICATION DYNAMIQUE DES OBJETS

N.B. Ce paragraphe peut être ignoré dans un premier temps.

Nous avons vu que la technique des fonctions virtuelles permettait de mettre en œuvre la ligature dynamique pour les fonctions concernées. Cependant, pour l'instant, cette technique peut vous apparaître comme une simple recette. La compréhension plus fine du mécanisme, et donc sa portée véritable, passe par la connaissance de la manière dont il est effectivement implanté. Bien que cette implémentation ne soit pas explicitement imposée par le langage, nous vous proposons de décrire ici la démarche couramment adoptée par les différents compilateurs existants.

Pour ce faire, nous allons considérer un exemple un peu plus général que le précédent, à savoir :

- une classe point comportant deux fonctions virtuelles :

```
class point
{     .....
      virtual void identifie () ;
      virtual void deplace (...) ;
      .....
} ;
```

- une classe pointcol, dérivée de point, ne redéfinissant que identifie :

```
class pointcol : public point
{     .....
      void identifie () ;
      .....
} ;
```

D'une manière générale, lorsqu'une classe comporte au moins une fonction virtuelle, le compilateur lui associe une table contenant les adresses de chacune des fonctions virtuelles correspondantes. Avec l'exemple cité, nous obtiendrons les deux tables suivantes :

- lors de la compilation de point :

Table de point

- lors de la compilation de pointcol :

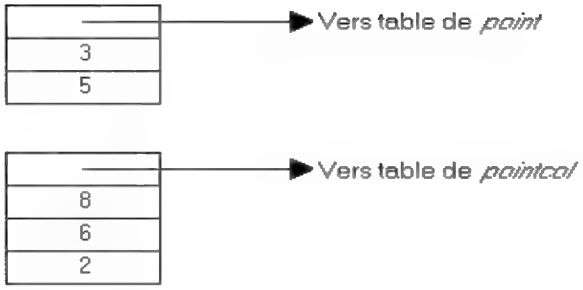
Table de pointcol

Notez qu'ici la seconde adresse de la table de pointcol est la même que pour la table de point, dans la mesure où la fonction deplace n'a pas été redéfinie.

D'autre part, tout objet d'une classe comportant au moins une fonction virtuelle se voit attribuer par le compilateur, outre l'emplacement mémoire nécessaire à ses membres donnée, un emplacement supplémentaire de type pointeur, contenant l'adresse de la table associée à sa classe. Par exemple, si nous déclarons (en supposant que nous disposons des constructeurs habituels) :

```
point p (3, 5) ;
pointcol pc (8, 6, 2) ;
```

nous obtiendrons :



On peut ainsi dire que ce pointeur, introduit dans chaque objet, représente l'information permettant d'identifier la classe de l'objet. C'est effectivement cette information qui est exploitée pour mettre en œuvre la ligature dynamique. Chaque appel d'une fonction virtuelle est traduit par le compilateur de la façon suivante :

- prélèvement dans l'objet de l'adresse de la table correspondante (quelle que soit la manière dont une fonction est appelée – directement ou par pointeur –, elle reçoit toujours l'adresse de l'objet en argument implicite),
- branchement à l'adresse figurant dans cette table à un rang donné. Notez bien que ce rang est parfaitement défini à la compilation : toutes les tables comporteront l'adresse de deplace[11], par exemple, en position 2. En revanche, c'est lors de l'exécution que sera effectué le "choix de la bonne table".

11 - Eventuellement, les tables de certaines classes pourront contenir plus d'adresses si elles introduisent de nouvelles fonctions virtuelles, mais celles qu'elles partagent avec leurs ascendantes occuperont toujours la même place et c'est là l'essentiel pour le bon déroulement des opérations.

XVI. LES FLOTS

Au cours des précédents chapitres, nous avons souvent été amené à écrire sur la "sortie standard". Pour ce faire, nous utilisions des instructions telles que :

```
cout << n ;
```

Cette dernière fait appel à l'opérateur << , en lui fournissant deux opérandes correspondant, d'une part au "flot de sortie" concerné (ici cout), d'autre part à l'expression dont on souhaite écrire la valeur (ici n).

De manière comparable, nous avons été amené à lire sur "l'entrée standard" en utilisant des instructions telles que :

```
cin >> x ;
```

Celle-ci fait appel à l'opérateur >> , en lui fournissant deux opérandes correspondant, d'une part au "flot d'entrée" concerné (ici cin), d'autre part à la "lvalue" dans laquelle on souhaite lire une information.

D'une manière générale, un flot peut être considéré comme un "canal" :

- recevant de l'information, dans le cas d'un flot de sortie,
- fournissant de l'information, dans le cas d'un flot d'entrée.

Les opérateurs << ou >> servent à assurer le transfert de l'information, ainsi que son éventuel "formatage".

Un flot peut être connecté à un périphérique ou à un fichier. Le flot prédéfini cout est, par convention, connecté à ce que l'on nomme la "sortie standard", correspondant au fichier prédéfini stdout du langage C. De même, le flot prédéfini cin est, par convention, connecté à ce que l'on nomme "l'entrée standard", correspondant au fichier prédéfini stdin du langage C. Généralement, l'entrée standard correspond au clavier et la sortie standard à l'écran[1]. Mais la plupart des implémentations permettent de "rediriger" l'entrée standard ou la sortie standard vers un fichier.

En dehors de ces flots prédéfinis[2], l'utilisateur peut définir lui-même d'autres flots qu'il pourra connecter à un fichier de son choix.

Si l'on y regarde de plus près, on peut dire qu'un flot est un **objet** d'une classe prédéfinie, à savoir :

---

1 - Toutefois, cela n'est pas imposé par le langage. On peut trouver, par exemple, des (rares) implémentations dans lesquelles la sortie standard correspond à une imprimante.

2 - Nous verrons qu'il en existe d'ailleurs deux autres : cerr et clog.

- **ostream** pour un flot de sortie,
- **istream** pour un flot d'entrée.

Chacune de ces deux classes surdéfinit les opérateurs << et >> pour les différents types de base. Leur emploi nécessite l'incorporation du fichier en-tête iostream.h[3].

Jusqu'ici, nous nous sommes contenté d'exploiter quelques-unes des possibilités des classes istream et ostream, en nous limitant, de plus, aux flots prédéfinis cin et cout. Ce chapitre va faire le point sur l'ensemble des possibilités d'entrées-sorties offertes par C++, telles qu'elles sont prévues depuis la version 2.0.

Nous adopterons la progression suivante :

- présentation générale des possibilités de la classe ostream : types de base acceptés, principales fonctions membre (put, write), exemples de formatage de l'information,
- présentation générale des possibilités de la classe istream : types de base acceptés, principales fonctions membre (get, getline, gcount, read...),
- gestion de ce que l'on nomme le "statut d'erreur d'un flot",
- possibilités de surdéfinition des opérateurs << et >> pour des types (classes) définis par l'utilisateur,
- étude détaillée des possibilités de formatage des informations, aussi bien en entrée qu'en sortie,
- connexion d'un flot à un fichier, et possibilités d'accès direct offertes dans ce cas,
- possibilités de "lecture ou d'écriture en mémoire", ce qui permettra de retrouver les facilités offertes par les fonctions fscanf et sprintf du langage C.

D'une manière générale, gardez bien présent à l'esprit, au cours de l'étude de ce chapitre, que tout ce qui sera dit dès le début, à propos des flots, s'appliquera sans restriction aucune à un flot quelconque et, donc, à un flot connecté à un fichier.

1. PRÉSENTATION GÉNÉRALE DE LA CLASSE OSTREAM

Après avoir précisé le rôle de l'opérateur << et rappelé les types de base pour lesquels l'opérateur << est surdéfini, nous verrons le rôle des deux fonctions membre put et write. Nous examinerons ensuite quelques exemples de formatage de l'information, ce qui nous permettra d'introduire l'importante notion de "manipulateur".

1.1 Surdéfinition de l'opérateur <<

Dans la classe ostream, l'opérateur << est donc surdéfini pour les différents types de base, sous la forme :

```
ostream & operator << (expression)
```

Il reçoit deux opérandes :

- la classe l'ayant appelé (argument implicite - this),
- une expression d'un type de base quelconque.

---

3 - Son extension dépend de l'implémentation (hxx, h...).

Son rôle consiste à "transmettre" la valeur de l'expression au flot concerné en la "formatant[4]" de façon appropriée. Considérons, par exemple, l'instruction :

```
cout << n ;
```

Si n contient la valeur 1234, le travail de l'opérateur << consistera à convertir la valeur (binaire) de n dans le système décimal et à envoyer au flot cout les caractères correspondant à chacun des chiffres ainsi obtenus (soit, ici, les quatre caractères : 1, 2, 3 et 4). Nous emploierons le mot "écriture" pour qualifier le rôle de cet opérateur ; sachez toutefois que ce terme n'est pas universellement répandu : notamment, on rencontre parfois "injection".

Par ailleurs, cet opérateur << fournit comme résultat la référence au flot concerné, après qu'il y a écrit l'information voulue. Cela permet de l'appliquer facilement plusieurs fois de suite, comme dans :

```
cout << "valeur : " << na << "\n" ;
```

Tous les types de base sont acceptés par l'opérateur <<[5] (soit par surdéfinition effective de l'opérateur, soit par le jeu des conversions implicites), y compris les types :

- char,
- char * : on obtient la chaîne située à l'adresse correspondante,
- pointeur sur un type quelconque : dans ce cas (du moins s'il ne s'agit pas de char *), on obtient la valeur du pointeur correspondant. Si l'on souhaite afficher la valeur d'un pointeur de type char * (et non plus la chaîne qu'il référence), il suffit de le convertir explicitement en void *.

1.2 Les flots prédéfinis

En plus de cout, il existe deux autres flots prédéfinis de classe ostream :

- cerr : flot de sortie connecté à la sortie standard d'erreur (stderr en C), sans "tampon"[6] intermédiaire,
- clog : flot de sortie connecté également à la sortie standard d'erreur, mais en utilisant un "tampon"[7] intermédiaire.

1.3 La fonction put

Il existe, dans la classe ostream, une fonction membre nommée put qui transmet au flot correspondant le caractère reçu en argument. Ainsi :

```
cout.put(c) ;
```

transmet au flot cout le caractère contenu dans c, comme le ferait :

```
cout << c ;
```

---

4 - Nous verrons qu'il est possible d'intervenir sur la manière dont est effectué ce formatage. D'autre part, dans certains cas, il pourra ne pas y avoir de formatage : c'est ce qui se produira, par exemple, lorsque l'on utilisera la fonction write.

5 - Du moins dans la version 2.0.

6 - "Buffer", en anglais. On parle parfois en "franglais", de sortie "non bufferisée".

7 - On parle parfois de sortie "bufferisée".

En fait, la fonction put était surtout indispensable dans les versions antérieures à la 2.0 pour pallier l'absence de surdéfinition de l'opérateur pour le type char.

La valeur de retour de put est le flot concerné, après qu'on y a écrit le caractère correspondant. Cela permet d'écrire, par exemple (c1, c2 et c3 étant de type char) :

```
cout.put(c1).put(c2).put(c3) ;
```

ce qui est équivalent à :

```
cout.put(c1) ;
cout.put(c2) ;
cout.put(c3) ;
```

1.4 La fonction write

Dans la classe ostream, la fonction membre write permet de transmettre au flot de sortie considéré une suite de caractères (octets) de longueur donnée. Par exemple, avec :

```
char t[] = "bonjour" ;
  ...
```

l'instruction :

```
cout.write (t, 4) ;
```

enverra sur le flot cout 4 caractères consécutifs à partir de l'adresse t, c'est-à-dire les caractères b, o, n et j.

Cette fonction peut sembler faire double emploi avec la transmission d'une chaîne à l'aide de l'opérateur <<. En fait, il faut tout d'abord noter que son comportement n'est pas identique puisque write ne fait pas intervenir de caractère de fin de chaîne (caractère nul) ; si un tel caractère apparaît dans la longueur prévue, il sera transmis, comme les autres, au flot de sortie. D'autre part, cette fonction ne réalise aucun formatage (nous verrons qu'avec <<, on peut agir sur le "gabarit" de l'information effectivement écrite sur le flot).

En fait, cette fonction s'avérera indispensable dès lors que l'on souhaitera transmettre une information sous une forme "brute" (on dit souvent "binaire"), sans qu'elle subisse la moindre modification. Bien entendu, cela n'a généralement guère d'intérêt dans le cas d'un écran ; en revanche, ce sera la seule façon de créer un fichier sous forme "binaire" (c'est-à-dire dans lequel les informations – quel que soit leur type – sont enregistrées telles qu'elles figurent en mémoire).

Comme put, la fonction write fournit en retour le flot concerné, après qu'on y a écrit l'information correspondante.

1.5 Quelques possibilités de formatage

Nous étudierons, dans le paragraphe 5, l'ensemble des possibilités de formatage de la classe ostream, ainsi d'ailleurs que celles de la classe istream. Cependant, dès maintenant, nous vous présentons deux exemples courants de formatage de l'information écrite sur un flot de sortie, ce qui nous permettra d'introduire la notion de "manipulateur".

a) Action sur la base de numération

Lorsque l'on écrit une valeur entière sur un flot de sortie, on peut choisir de l'exprimer dans l'une des bases suivantes :

- 10 : décimal (il s'agit de la valeur par défaut),
- 16 : hexadécimal,
- 8 : octal.

Voici un exemple de programme dans lequel nous écrivons dans différentes bases la valeur de la même variable (n).

```
#include <iostream.h>
main()
{  int n = 12000 ;
   cout << "par défaut     : "        << n << "\n" ;
   cout << "en hexadécimal : " << hex << n << "\n" ;
   cout << "en décimal     : " << dec << n << "\n" ;
   cout << "en octal       : " << oct << n << "\n" ;
   cout << "et en ????     : "        << n << "\n" ;
}
```

```
par défaut     : 12000
en hexadécimal : 2ee0
en décimal     : 12000
en octal       : 27340
et en ????     : 27340
```

Action sur la base de numération des valeurs écrites sur cout

Les symboles hex, dec et oct se nomment des **manipulateurs**. Il s'agit, en fait, d'opérateurs prédéfinis, à un seul opérande de type flot, et fournissant en retour le même flot, après qu'ils y ont opéré une certaine action ("manipulation"). Ici, cette action consiste précisément à modifier la valeur de la base de numération (cette information étant, en fait, mémorisée dans la classe ostream[8]). Notez bien que la valeur de la base reste la même, tant qu'on ne la modifie pas (par un manipulateur), et cela quelles que soient les informations transmises au flot (entiers, caractères, flottants...).

Nous verrons, dans le paragraphe 5, qu'il existe beaucoup d'autres manipulateurs et nous en ferons une étude complète.

b) Action sur le gabarit de l'information écrite

Voyez cet exemple de programme qui montre comment agir sur la largeur (gabarit) suivant laquelle l'information est écrite :

```
#include <iostream.h>
#include <iomanip.h>
main()
```

8 - En toute rigueur, de la classe ios, dont dérivent les classes ostream et istream.

```
{
   int n = 12345 ;
   int i ;
   for (i=0 ; i<15 ; i++)
      cout << setw(2) << i << " : "<< setw(i) << n << "\n" ;
```

```
 0 : 12345
 1 : 12345
 2 : 12345
 3 : 12345
 4 : 12345
 5 : 12345
 6 :  12345
 7 :   12345
 8 :    12345
 9 :     12345
10 :      12345
11 :       12345
12 :        12345
13 :         12345
14 :          12345
```

Action sur le gabarit de l'information écrite sur cout

Ici encore, nous faisons appel à un manipulateur (setw). Celui-ci est cependant un peu plus complexe que les précédents (hex, oct ou dec) puisqu'il comporte un "paramètre" représentant, dans ce cas, le gabarit souhaité. On parle alors de "manipulateur paramétrique". Nous verrons qu'il existe beaucoup d'autres manipulateurs paramétriques ; leur emploi nécessite absolument l'incorporation du fichier en-tête iomanip.h.

En ce qui concerne setw, sachez que ce manipulateur définit uniquement le gabarit de la **prochaine information à écrire**. Si l'on ne fait pas à nouveau appel à setw pour les informations suivantes, celles-ci seront écrites suivant les conventions habituelles, à savoir en utilisant l'emplacement minimum nécessaire pour les écrire (2 caractères pour la valeur 24, 5 caractères pour la valeur -2345, 7 caractères pour la chaîne "bonjour"...). D'autre part, si la valeur fournie à setw est insuffisante à l'écriture de la valeur suivante, cette dernière sera écrite suivant les conventions habituelles (elle ne sera donc pas tronquée).

2. PRÉSENTATION GÉNÉRALE DE LA CLASSE ISTREAM

De manière comparable à ce que nous avons fait pour la classe ostream, nous commencerons par préciser le rôle de l'opérateur > > . Nous verrons ensuite le rôle des différentes fonctions membre de la classe istream (get, getline, gcount, read...). Nous terminerons sur un exemple de formatage de l'information.

2.1 Surdéfinition de l'opérateur > >

Dans la classe istream, l'opérateur > > est surdéfini pour tous les types de base, y compris char *[9], sous la forme :

```
istream & operator >> ( & type_de_base )
```

Il reçoit deux opérandes :

- la classe l'ayant appelé (argument implicite - this),
- une "lvalue" d'un type de base quelconque.

Son rôle consiste à "extraire" du flot concerné les caractères nécessaires pour former une valeur du type de base voulu en réalisant une opération inverse du formatage opéré par l'opérateur < < .

Il fournit comme résultat la référence au flot concerné, après qu'il en a extrait l'information voulue. Cela permet de l'appliquer plusieurs fois de suite, comme dans :

```
cin >> n >> p >> x ;
```

Les "espaces_blancs[10]" servent de délimiteurs. Rappelons que l'on range dans cette catégorie les caractères suivants : espace, tabulation horizontale (\t), tabulation verticale (\v), fin de ligne (\n), retour chariot (\r) et changement de page (\f).

Bien entendu, les délimiteurs ne peuvent pas être lus, en tant que caractères eux-mêmes. Par exemple, la répétition de l'instruction (c étant supposé de type char) :

```
cin >> c ;
```

appliquée à un flot contenant ce texte :

```
b o
n    j

our
```

conduira à ne prendre en compte que les 7 caractères : b, o, n, j, o, u et r.

Nous verrons ci-dessous comment procéder (avec la fonction get) pour accéder à tous les caractères d'un flot, délimiteurs compris.

D'autre part, lorsqu'on lit sur un flot une information à destination d'une chaîne de caractères (char *) :

- ce sont, là encore, les "espaces_blancs" qui servent de délimiteurs. Il n'est donc pas possible de lire en une seule fois une chaîne contenant, par exemple, un espace, telle que :

  ```
  bonjour mademoiselle
  ```

  Notez que, dans ce cas, il ne sert à rien de la placer entre guillemets, comme ceci :

  ```
  "bonjour mademoiselle"
  ```

  En effet, la première chaîne lue serait alors :

  ```
  "bonjour
  ```

9 - Mais pas, a priori, pour les types pointeurs.

10 - De "white spaces", en anglais.

Nous verrons, un peu plus loin, comment la fonction getline fournit une solution agréable au problème que nous venons d'évoquer ;

- l'information rangée en mémoire est complétée par un caractère nul de fin de chaîne (\0). Ainsi, pour lire une chaîne de n caractères, il faut prévoir un emplacement de n+1 caractères.

2.2 La fonction get

La fonction :

```
istream & get (char &)
```

permet d'extraire un caractère d'un flot d'entrée et de le ranger dans la variable (de type char) qu'on lui fournit en argument. Tout comme put, cette fonction fournit en retour la référence au flot concerné, après qu'on en a extrait le caractère voulu.

Contrairement à l'opérateur >>, la fonction get peut lire n'importe quel caractère, délimiteurs compris. Ainsi, en l'appliquant à un flot contenant ce texte :

```
b o
n    j

our
```

elle conduira à prendre en compte 16 caractères : b, espace, o, \n, n, espace, espace, espace, espace, j, \n, \n, o, u, r et \n.

Il existe une autre fonction get (il y a donc surdéfinition), de la forme :

```
int get ()
```

Celle-ci permet, elle aussi, d'extraire un caractère d'un flot d'entrée, mais, cette fois, elle le fournit, comme valeur de retour, sous forme d'un **entier**. Elle est ainsi en mesure de fournir une valeur spéciale EOF (en général -1) lorsque la fin de fichier a été rencontrée sur le flot correspondant[11].

Remarque :

Nous verrons, dans le paragraphe 3 consacré au "statut d'erreur" d'un flot, qu'il est possible de considérer un flot comme une "valeur logique" (vrai ou faux) et, par suite, d'écrire des instructions telles que :

```
char c ;
  ...
while ( cin.get(c) )       // recopie le flot cin
   cout.put (c) ;          // sur le flot cout
                           // arrêt quand eof car alors (cin) = 0
```

Celles-ci sont équivalentes à :

```
int c ;
  ...
```

11 - C'est ce qui justifie que sa valeur de retour soit de type int et non char.

```
while ( ( c = cin.get() ) != EOF )
   cout.put (c) ;
```

2.3 Les fonctions getline et gcount

Ces deux fonctions facilitent la lecture des chaînes de caractères ou, plus généralement, la lecture d'une suite de caractères quelconques, terminée par un caractère connu (et non présent dans la chaîne en question).

L'en-tête de la fonction getline se présente sous la forme :

```
istream & getline (char * ch, int taille, char delim = '\n' )
```

Cette fonction lit des caractères sur le flot l'ayant appelé et les place dans l'emplacement d'adresse ch. Elle s'interrompt lorsqu'une des deux conditions suivantes est satisfaite :

- le caractère délimiteur delim a été trouvé : dans ce cas, ce caractère n'est pas recopié en mémoire.
- taille - 1 caractères ont été lus.

Dans tous les cas, cette fonction ajoute un caractère nul de fin de chaîne, à la suite des caractères lus.

Notez que le caractère délimiteur possède une valeur par défaut (\n) bien adaptée à la lecture de lignes de texte.

Quant à la fonction gcount, elle fournit le nombre de caractères effectivement lus lors du dernier appel de getline. Le caractère délimiteur, pas plus que celui placé à la fin de la chaîne ne sont comptés ; autrement dit, gcount fournit la longueur effective de la chaîne rangée en mémoire par getline.

Voici, à titre d'exemple, une séquence classique d'instructions permettant de traiter les différentes lignes du flot cin (éventuellement redirigé vers un fichier) :

```
const LG_LIG = 120 ;         // longueur maxi d'une ligne de texte
   ...
char ch [LG_LIG+1] ;         // pour lire une ligne
int lg ;                     // longueur courante d'une ligne
   ...
while ( cin.getline (ch, LG_LIG) )
   { lg = cin.gcount () ;
       // traitement d'une ligne de lg caractères
         ...
   }
```

2.4 La fonction read

La fonction read permet de lire sur le flot d'entrée considéré une suite de caractères (octets) de longueur donnée. Par exemple, avec :

```
char t[10] ;
```

l'instruction :

```
cin.read (t, 5) ;
```

lira sur cin 5 caractères et les rangera à partir de l'adresse t.

Ici encore, cette fonction peut sembler faire double emploi soit avec la lecture d'une chaîne avec l'opérateur >>, soit avec la fonction getline ; toutefois, read ne nécessite ni séparateur ni caractère délimiteur particulier.

En fait, nous verrons que cette fonction s'avérera indispensable dès lors que l'on souhaitera accéder à des fichiers sous forme "binaire", c'est-à-dire en recopiant en mémoire les informations telles qu'elles figurent dans le fichier. La fonction read jouera le rôle symétrique de la fonction write.

2.5 Quelques autres fonctions

Il existe également, dans la classe istream, deux fonctions membre, à caractère utilitaire :

putback (char c) pour renvoyer dans le flot concerné un caractère donné,

peek () qui fournit le prochain caractère disponible sur le flot concerné, mais sans l'extraire du flot (il sera donc à nouveau obtenu lors d'une prochaine lecture sur le flot).

Remarque :

En toute rigueur, il existe également une classe iostream, héritant à la fois de istream et de ostream. Celle-ci permet de réaliser des entrées-sorties "bidirectionnelles".

3. STATUT D'ERREUR D'UN FLOT

A chaque flot (d'entrée ou de sortie) est associé un ensemble de bits d'un entier, formant ce que l'on nomme le "statut d'erreur" du flot. Il permet de rendre compte du bon ou du mauvais déroulement des opérations sur le flot. Nous allons tout d'abord voir quelle est précisément la signification de ces différents bits (au nombre de 4). Puis nous apprendrons comment en connaître la valeur et, le cas échéant, la modifier. Enfin, nous verrons comment la surdéfinition des opérateurs () et ! permet de simplifier l'utilisation d'un flot.

3.1 Les bits d'erreur

La position des différents bits d'erreur, au sein d'un entier, est définie par des constantes déclarées dans la classe ios dont dérivent les deux classes istream et ostream. Chacune de ces constantes correspond à la valeur prise par l'entier en question lorsque le bit correspondant (et lui seul) est "activé" (à 1). Il s'agit de :

eofbit : fin de fichier ; ce bit est activé si la fin de fichier a été atteinte, autrement dit si le flot correspondant n'a plus aucun caractère disponible.

failbit : ce bit est activé lorsque la prochaine opération d'entrée-sortie ne peut aboutir,

badbit : ce bit est activé lorsque le flot est dans un état irrécupérable.

La différence entre badbit et failbit n'existe que pour les flots d'entrée ; lorsque failbit est activé, aucune information n'a été réellement perdue sur le flot ; il n'en va plus de même lorsque badbit est activé.

De plus, il existe une constante goodbit (valant, en fait 0), correspondant à la valeur que doit avoir le statut d'erreur lorsqu'aucun de ses bits n'est activé.

On peut dire qu'une opération d'entrée-sortie a réussi lorsque l'un des bits goodbit ou eofbit est activé. De même, on peut dire que la prochaine opération d'entrée-sortie ne pourra aboutir que si goodbit est activé (mais il n'est pas encore certain qu'elle réussisse!).

Lorsqu'un flot est dans un état d'erreur, aucune opération ne peut aboutir tant que :

- la condition d'erreur n'a pas été corrigée (ce qui va de soi !),
- et que le bit d'erreur corrrespondant n'a pas été remis à zéro : nous allons voir, ci-dessous, qu'il existe des fonctions permettant d'agir sur ces bits d'erreur.

3.2 Action concernant les bits d'erreur

Il existe deux catégories de fonctions :

- celles qui permettent de connaître le statut d'erreur d'un flot, c'est-à-dire, en fait, la valeur de ses différents bits d'erreur,
- celles qui permettent de modifier la valeur de certains de ces bits d'erreur.

a) Accès aux bits d'erreur

D'une part, il existe 5 fonctions membre (de ios[12]) :

eof () : fournit la valeur vrai (1) si la fin de fichier a été rencontrée, c'est-à-dire si le bit eofbit est activé.

bad () : fournit la valeur vrai (1) si le flot est altéré, c'est-à-dire si le bit badbit est activé.

fail () : fournit la valeur vrai (1) si le bit failbit est activé,

good () : fournit la valeur vrai (1) si aucune des trois fonctions précédentes n'a la valeur vrai, c'est-à-dire si aucun des bits du statut d'erreur n'est activé.

D'autre part, la fonction membre[13] rdstate () fournit en retour un entier correspondant à la valeur du statut d'erreur.

b) Modification du statut d'erreur

La fonction membre clear d'en-tête :

void clear (int i= 0)

active les bits d'erreur correspondant à la valeur fournie en argument. En général, on définit la valeur de cet argument en utilisant les constantes prédéfinies de la classe ios.

Par exemple, si fl désigne un flot, l'instruction :

```
fl.clear (ios::badbit) ;
```

activera le bit badbit du statut d'erreur du flot fl et mettra tous les autres bits à zéro.

---

12 - Donc de istream et de ostream, par héritage.

13 - Désormais, nous ne préciserons plus qu'il s'agit d'un membre de ios, dont héritent istream et ostream.

Si l'on souhaite activer ce bit, sans modifier les autres, il suffit de faire appel à rdstate, en procédant ainsi :

```
fl.clear (ios::badbit | fl.rdstate () ) ;
```

Remarque :

Lorsque vous surdéfinirez les opérateurs << et >> pour vos propres types (classes), il sera pratique de pouvoir activer les bits d'erreur, en compte rendu du déroulement de l'opération.

3.3 Surdéfinition des opérateurs () et !

Comme nous l'avons déjà évoqué dans la remarque du paragraphe 2.2, il est possible de "tester" un flot en le considérant comme une valeur logique (vrai ou faux). Cela est réalisé grâce à la surdéfinition, dans la classe ios des opérateurs () et !.

Plus précisément, l'opérateur () est surdéfini de manière que, si fl désigne un flot :

(fl)

- prenne une valeur non nulle[14] (vrai), si aucun des bits d'erreur n'est activé, c'est-à-dire si good () a la valeur vrai.
- prenne une valeur nulle (faux), dans le cas contraire, c'est-à-dire si good () a la valeur faux.

Ainsi :

```
if (fl) ...
```

peut remplacer :

```
if (fl.good () ) ...
```

De même, l'opérateur ! est surdéfini, de manière que, si fl désigne un flot :

! fl

- prenne une valeur nulle (faux) si un des bits d'erreur est activé, c'est-à-dire si good () a la valeur faux,
- prenne une valeur non nulle (vrai) si aucun des bits d'erreur n'est activé, c'est-à-dire si good () a la valeur vrai.

Ainsi :

```
if (! flot ) ...
```

peut remplacer :

```
if (! flot.good () ) ...
```

[14] - Sa valeur exacte n'est pas précisée et elle n'a donc pas de signification particulière.

4. SURDÉFINITION DES OPÉRATEURS < < ET > > POUR LES TYPES DÉFINIS PAR L'UTILISATEUR

Comme nous l'avons déjà dit, les opérateurs < < et > > peuvent être redéfinis par l'utilisateur pour des types classes qu'il a lui-même créés. Nous allons d'abord examiner la démarche à employer pour réaliser cette surdéfinition, avant d'en voir un exemple d'application.

4.1 La démarche

Les deux opérateurs < < et > >, déjà surdéfinis au sein des classes istream et ostream pour les différents types de base, peuvent être surdéfinis pour n'importe quel type classe créé par l'utilisateur.

Pour ce faire, il suffit de tenir compte des remarques suivantes.

a) Ces opérateurs doivent recevoir un flot en premier argument, ce qui empêche d'en faire une fonction membre de la classe concernée (notez qu'on ne peut plus, comme dans le cas des types de base, en faire une fonction membre de la classe istream ou ostream, dans la mesure où l'utilisateur ne peut plus modifier ces classes qui lui sont fournies avec C++).

   Il s'agira donc de fonctions indépendantes ou amies de la classe concernée et ayant un prototype de la forme :

```
ostream & operator << (ostream &, expression_de_type_classe)
```

   ou :

```
istream & operator >> (ostream &, & type_classe)
```

b) La valeur de retour sera obligatoirement la référence au flot concerné (reçu en premier argument).

On peut dire que toutes les surdéfinitions de < < se feront suivant ce "canevas" :

```
ostream & operator << (ostream & sortie, type_classe objet[15])
{
     // Envoi sur le flot sortie des membres de objet en utilisant
     // les possibilités classiques de << pour les types de base
     // c'est-à-dire des instructions de la forme :
     //     sortie << ..... ;
   return sortie ;
}
```

De même, toutes les surdéfinitions de > > se feront suivant ce "canevas" :

```
istream & operator >> (istream & entree, type_classe & objet)
{
     // Lecture des informations correspondant aux différents membres de objet
     // en utilisant les possibilités classiques de >> pour les types de base
     // c'est-à-dire des instructions de la forme :
     //     entree >> ..... ;
   return entree ;
}
```

15 - Ici, la transmission peut se faire par valeur ou par référence.

Remarque :

Dans le cas de la surdéfinition de > > (flot d'entrée), il sera souvent utile de s'assurer que l'information lue répond à certaines exigences et d'agir en conséquence sur l'état du flot. Nous en verrons précisément un exemple dans le paragraphe suivant.

4.2 Exemple

Voici un programme dans lequel nous avons surdéfini les opérateurs < < et > > pour le type point que nous avons souvent rencontré dans les précédents chapitres :

```
class point
{    int x , y ;
   .....
 ;
```

Nous supposerons qu'une "valeur de type point" se présente toujours (aussi bien en lecture qu'en écriture) sous la forme :

< entier, entier >

avec éventuellement des séparateurs "espaces_blancs" supplémentaires, de part et d'autre des valeurs entières.

```
#include <iostream.h>
class point
{  int x, y ;
  public :
   point (int abs=0, int ord=0)
     { x = abs ; y = ord ; }
   int abscisse () { return x ; }
   friend ostream & operator << (ostream &, point) ;
   friend istream & operator >> (istream &, point &) ;
} ;
ostream & operator << (ostream & sortie, point p)
{
  sortie << "<" << p.x << "," << p.y << ">" ;
  return sortie ;
}
istream & operator >> (istream & entree, point & p)
{  char c = '\0' ;
   float x, y ;
   int ok = 1 ;
   entree >> c ;
   if (c != '<') ok = 0 ;
      else
       { entree >> x >> c ;
         if (c != ',') ok = 0 ;
            else
             { entree >> y >> c ;
               if (c != '>') ok = 0 ;
             }
```

```
        }
   if (ok) { p.x = x ; p.y = y ; }         // on n'affecte à p que si tout est
OK
      else entree.clear (ios::badbit | entree.rdstate () ) ;
   return entree ;
}

main()
{ char ligne [121] ;
  point a(2,3), b ;
  cout << "point a : " << a << "  point b : " << b << "\n" ;
  do
   { cout << "donnez un point : " ;
     if (cin >> a)  cout << "merci pour le point : " << a << "\n" ;
                    else { cout << "** information incorrecte \n" ;
                           cin.clear () ;
                           cin.getline (ligne, 120, '\n') ;
                         }
   }
  while ( a.abscisse () ) ;
}
```

```
point a : <2,3>  point b : <0,0>
donnez un point : 4,5
** information incorrecte
donnez un point : <4,5<
** information incorrecte
donnez un point : <4,5>
merci pour le point : <4,5>
donnez un point : < 8,   9     >
merci pour le point : <8,9>
donnez un point : bof
** information incorrecte
donnez un point : <0,0>
merci pour le point : <0,0>
```

Surdéfinition de l'opérateur << pour la classe point

Voyez comment, dans la surdéfinition de >>, nous avons pris soin de lire tout d'abord toutes les informations relatives à un point dans des variables locales. Ce n'est que lorsque tout s'est bien déroulé que nous transférons les valeurs ainsi lues dans le point concerné. Cela évite, par exemple en cas d'information incomplète, de modifier l'une des composantes du point sans modifier l'autre, ou encore de modifier les deux composantes, alors que le caractère > de fin n'a pas été trouvé.

Si nous ne prenions pas soin d'activer le bit badbit lorsque l'on ne trouve pas l'un des caractères < ou >, il serait impossible (à l'utilisateur) de savoir que la lecture s'est mal déroulée.

Notez que dans la fonction main, en cas d'erreur sur cin, nous commençons par remettre à zéro l'état du flot avant d'utiliser getline pour "sauter" les informations qui risquent de ne pas avoir pu être exploitées.

5. GESTION DU FORMATAGE

Nous avons vu quelques possibilités d'action sur le formatage des informations, aussi bien pour un flot d'entrée que pour un flot de sortie. Nous allons ici étudier en détail la démarche adoptée par C++ pour gérer ce formatage.

Chaque flot, c'est-à-dire chaque objet de classe istream ou ostream, conserve en permanence un ensemble d'informations[16] (indicateurs) spécifiant quel est, à un moment donné, son "statut de formatage". Cette façon de procéder est fortement différente de celle employée par les fonctions C telles que printf ou scanf ; dans ces dernières, en effet, on fournissait, pour chaque opération d'entrée-sortie, les indications de formatage appropriées (sous forme d'un "format" composé, entre autres, d'une succession de "codes de format").

Un des avantages les plus éminents de la méthode employée par C++ est qu'elle permet éventuellement à l'utilisateur d'ignorer totalement cet aspect formatage, tant qu'il se contente d'un comportement par défaut (ce qui est loin d'être le cas en C où la moindre entrée-sortie nécessite obligatoirement l'emploi d'un format).

Un autre avantage de la méthode est de permettre à celui qui le souhaite de définir, une fois pour toutes, un format approprié à une application donnée et de ne plus avoir à s'en soucier par la suite.

Comme nous l'avons fait pour le statut d'erreur d'un flot, nous commencerons par étudier les différents éléments composant le "statut de formatage" d'un flot avant de voir comment on peut le connaître d'une part, le modifier d'autre part.

5.1 Le statut de formatage d'un flot

Le statut de formatage d'un flot comporte essentiellement :

- un **mot d'état**, dans lequel chaque bit est associé à une signification particulière. On peut dire qu'on y trouve, en quelque sorte, toutes les indications de formatage de la forme vrai/faux[17].
- les valeurs numériques précisant les valeurs courantes suivantes :
  - le "gabarit" : il s'agit de la valeur fournie a setw ; rappelons qu'elle "retombe" à zéro (qui signifie : gabarit standard), après le transfert (lecture ou écriture) d'une information.
  - la "précision" numérique : il s'agit du nombre de chiffres affichés après le point décimal dans le cas de notation "flottante" et du nombre de chiffres significatifs, dans le cas de notation "exponentielle".
  - le caractère "de remplissage", c'est-à-dire le caractère employé pour compléter un gabarit, dans le cas où l'on n'utilise pas le gabarit par défaut (par défaut, ce caractère de remplissage est un espace).

5.2 Description du mot d'état du statut de formatage

De manière comparable à ce qui se passait pour le statut d'erreur d'un flot, le mot d'état du statut de formatage est formé d'un entier, dans lequel chaque bit est repéré par une constante prédéfinie dans la classe

16 - En toute rigueur, cette information est prévue dans la classe ios dont dérivent les classes istream et ostream.

17 - On retrouve là le même mécanisme que pour l'entier contenant le statut d'erreur d'un flot. Mais, comme nous le voyons ci-après, le statut de formatage d'un flot comporte, quant à lui, d'autres types d'informations que ces indications "binaires".

ios. Chacune de ces constantes correspond à la valeur prise par cet entier lorsque le bit correspondant (et lui seul) est "activé" (à 1). Ici encore, la valeur de chacune de ces constantes peut servir :

- soit à "identifier" le bit correspondant, au sein du mot d'état,
- soit à fabriquer directement un mot d'état.

De plus, certains "champs de bits" (au nombre de trois) sont définis au sein de ce même mot ; nous verrons qu'ils facilitent, dans le cas de certaines fonctions membre, la manipulation d'un des bits d'un champ (on peut "citer" le bit à modifier dans un champ, sans avoir à se préoccuper de la valeur des bits des autres champs).

Voici la liste des différentes constantes, accompagnées, le cas échéant, du nom du champ de bit correspondant.

NOM DE CHAMP (s'il existe)	NOM DU BIT	SIGNIFICATION (quand activé)
	ios::skipws	saut des "espaces blancs" (en entrée)
ios::adjustfield	ios::left	cadrage à gauche (en sortie)
	ios::right	cadrage à droite (en sortie)
	ios::internal	remplissage après signe ou base
ios::basefield	ios::dec	conversion décimale
	ios::oct	conversion octale
	ios::hex	conversion hexadécimale
	ios::showbase	affichage indicateur de base (en sortie)
	ios::showpoint	affichage point décimal (en sortie)
	ios::uppercase	affichage caractères hexadécimaux en majuscules (en sortie)
	ios::showpos	affichage nombres positifs précédés du signe + (en sortie)
ios::floatfield	ios::scientific	notation "scientifique"
	ios::fixed	notation "point fixe"

ios::unitbuf	vide les tampons après chaque écriture
ios::stdio	vide les tampons après chaque écriture sur stdout ou stderr

Le mot d'état du statut de formatage

Au sein de chacun des trois champs de bits (adjustfield, basefield, floatfield), un seul des bits doit être actif. S'il n'en va pas ainsi, C++ lève l'ambiguïté en prévoyant un comportement par défaut (right, dec, scientific).

5.3 Action sur le statut de formatage

Les exemples des paragraphes 1 et 2 vous ont présenté la notion de manipulateur (paramétrique ou non). Comme vous vous en doutez, ces manipulateurs permettent effectivement d'agir sur le statut de formatage. Mais il existe également d'autres façons d'agir sur ce statut, en utilisant des fonctions membre des classes istream ou ostream. Ces dernières sont généralement redondantes par rapport aux manipulateurs paramétriques (nous verrons toutefois qu'il existe des fonctions membre ne comportant aucun équivalent sous forme de manipulateur)

Suivant le cas, l'action portera sur le mot d'état ou sur les valeurs numériques (gabarit, précision, caractère de remplissage). En outre, on peut agir globalement sur le mot d'état. Nous verrons que certaines fonctions membre permettront notamment de le "sauvegarder" pour pouvoir le "restaurer" ultérieurement (ce qu'aucun manipulateur ne permet) ; les valeurs numériques, quant à elles, ne peuvent être accédées globalement et doivent donc, le cas échéant, faire l'objet de sauvegardes individuelles.

a) Les manipulateurs non paramétriques

Ce sont donc des opérateurs qui s'utilisent ainsi :

```
flot << manipulateur
```

pour un flot de sortie, ou ainsi :

```
flot >> manipulateur
```

pour un flot d'entrée.

Ils fournissent comme résultat le flot obtenu après leur action, ce qui permet de les traiter de la même manière que les informations à transmettre. En particulier, ils permettent, eux aussi, d'appliquer plusieurs fois de suite les opérateurs << ou >>.

Voici la liste de ces manipulateurs :

MANIPULATEUR	UTILISATION	ACTION

dec	Entrée/Sortie	Active le bit de conversion décimale
hex	Entrée/Sortie	Active le bit de conversion hexadécimale
oct	Entrée/Sortie	Active le bit de conversion octale
left/base/internal	Sortie	Active le bit correspondant
scientific/fixed	Sortie	Active le bit correspondant
showbase/noshowbase	Sortie	Active/désactive le bit correspondant
showpoint/noshowpoint	Sortie	Active/désactive le bit correspondant
showpos/noshowpos	Sortie	Active/désactive le bit correspondant
skipws/noskipws	Entrée	Active/désactive le bit correspondant
uppercase/nouppercase	Sortie	Active/désactive le bit correspondant
ws	Entrée	Active le bit de saut des caractères "espaces blancs"
endl	Sortie	Insère un saut de ligne et vide le tampon
ends	Sortie	Insère un caractère de fin de chaîne (\0)
flush	Sortie	Vide le tampon

Les manipulateurs non paramétriques

b) Les manipulateurs paramétriques

Ce sont donc également des manipulateurs, c'est-à-dire des opérateurs agissant sur un flot et fournissant en retour le flot après modification. Mais, cette fois, ils comportent un paramètre qui leur est fourni sous forme d'un argument entre parenthèses. En fait, ces manipulateurs paramétriques sont des fonctions dont l'en-tête est de la forme :

```
istream & manipulateur (argument)
```

ou :

```
ostream & manipulateur (argument)
```

Ils s'emploient comme les manipulateurs non paramétriques, avec toutefois cette différence qu'ils nécessitent l'inclusion du fichier iomanip.hpp[18].

Voici la liste de ces manipulateurs paramétriques :

MANIPULATEUR	UTILISATION	ROLE
setbase (int)	Entrée/Sortie	Définit la base de conversion
resetiosflags (long) par	Entrée/Sortie	Remet à zéro tous les bits désignés l'argument (sans modifier les autres)
setiosflags (long)	Entrée/Sortie	Active tous les bits spécifiés par l'argument (sans modifier les autres)
setfill (int)	Entrée/Sortie	Définit le caractère de remplissage
setprecision (int)	Entrée/Sortie	Définit la précision des nombres flottants
setw (int)	Entrée/Sortie	Définit le gabarit

Les manipulateurs paramétriques

Notez bien que les manipulateurs resetiosflags et setiosflags agissent sur **tous** les bits spécifiés par leur argument.

c) Les fonctions membre

Dans les classes istream et ostream, il existe 4 fonctions membre que nous n'avons pas encore rencontrées : setf, fill, precision et width.

setf

Cette fonction permet de modifier le mot d'état de formatage. Elle est en fait surdéfinie. Il existe deux versions :

long setf (long)

Son appel active les bits spécifiés par son argument. On obtient, en retour, l'ancienne valeur du mot d'état de formatage.

18 - L'extension peut varier suivant l'implémentation (hxx, h...).

Notez bien que, comme le manipulateur setiosflags, cette fonction ne modifie pas les autres bits. Ainsi, en supposant que flot est un flot, avec :

```
flot.setf (ios::oct)
```

on active le bit ios:oct, alors qu'un des autres bits ios:dec ou ios:hex est peut être activé. Comme nous allons le voir ci-dessous, la deuxième forme de setf se révèle plus pratique dans ce cas.

long setf (long, long)

Son appel active les bits spécifiés par le premier argument, au sein seulement du champ de bits défini par le second argument. Par exemple, si flot désigne un flot :

```
flot.setf (ios::oct, ios::basefield)
```

active le bit ios:oct en désactivant les autres bits du champ ios:basefield.

Cette version de setf fournit en retour l'ancienne valeur du **champ de bits** concerné. Cela permet d'éventuelles sauvegardes pour des restaurations ultérieures. Par exemple, si flot est un flot, avec :

```
base_a = flot.setf (ios::hex, ios::basefield) ;
```

vous passez en notation hexadécimale. Pour revenir à l'ancienne notation, quelle qu'elle soit, il vous suffira de procéder ainsi :

```
flot.setf (base_a, ios::basefield) ;
```

fill

Cette fonction permet d'agir sur le caractère de remplissage. Elle est également surdéfinie. Il existe deux versions :

char fill ()

Cette version fournit comme valeur de retour l'actuel caractère de remplissage.

char fill (char)

Cette version donne au caractère de remplissage la valeur spécifiée par son argument et fournit en retour l'ancienne valeur. Si flot est un flot de sortie, on peut, par exemple, imposer temporairement le caractère * comme caractère de remplissage, puis retrouver l'ancien caractère, quel qu'il soit, en procédant ainsi :

```
char car_a ;
 ....
car_a = fill ('*') ;        // caractère de remplissage = '*'
 ....
fill (car_a) ;              // retour à l'ancien caractère de remplissage
```

precision

Cette fonction permet d'agir sur la précision numérique. Elle est également surdéfinie. Il en existe deux versions.

int precision ()

Cette version fournit comme valeur de retour la valeur actuelle de la précision numérique.

int precision (int)

Cette version donne à la précision numérique, la valeur spécifiée par son argument et fournit en retour l'ancienne valeur. Si flot est un flot de sortie, on peut, par exemple, imposer temporairement une certaine précision (ici prec) puis revenir à l'ancienne précision, quelle qu'elle soit, en procédant ainsi :

```
int prec_a, prec ;
   .....
prec_a = flot.precision (prec) ;     // on impose la précision définie par
prec
   .....
flot.precision (prec_a) ;            // on revient à l'ancienne précision
```

width

Cette fonction permet d'agir sur le "gabarit". Elle est également surdéfinie. Il en existe deux versions.

int width()

Cette version fournit comme valeur de retour la valeur actuelle du gabarit.

int width(int)

Cette version donne au gabarit la valeur spécifiée par son argument et fournit en retour l'ancienne valeur. Si flot est un flot de sortie, on peut par exemple, imposer temporairement un certain gabarit (ici gab) puis revenir à l'ancien gabarit, quel qu'il soit en procédant ainsi :

```
int gab_a, gab ;
   .....
gab_a = flot.width (gab) ;     // on impose un gabarit défini par gab
   .....
flot.width (gab_a) ;           // on revient à l'ancien gabarit
```

6. CONNEXION D'UN FLOT À UN FICHIER

Jusqu'ici, soit nous avons parlé des flots prédéfinis (cin et cout), soit nous vous avons donné des informations s'appliquant à un flot quelconque (paragraphes 3 et 5), mais sans vous dire comment ce flot pourrait être associé à un fichier. Ce paragraphe va d'une part vous montrer comment y parvenir, d'autre part examiner les possibilités d'accès direct dont on peut alors bénéficier.

6.1 Connexion d'un flot de sortie à un fichier

Pour associer un flot de sortie à un fichier, il suffit en fait de créer un objet de type ofstream, classe dérivant de ostream. L'emploi de cette nouvelle classe nécessite d'inclure un fichier en-tête nommé fstream.h[19], en plus du fichier iostream.h.

Le constructeur de la classe ofstream nécessite deux arguments :

- le nom du fichier concerné (sous forme d'une chaîne de caractères),
- un mode d'ouverture défini par une constante entière : la classe ios comporte, là encore, un certain nombre de constantes prédéfinies (nous les passerons toutes en revue, dans le paragraphe 7.4).

Voici un exemple de déclaration d'un objet (nommé ici sortie) du type ofstream :

```
ofstream sortie ("truc.dat", ios::out) ;
```

L'objet sortie sera donc associé au fichier nommé truc.dat, après qu'il a été ouvert en écriture.

Une fois construit un objet de classe ofstream, l'écriture dans le fichier qui lui est associé peut se faire comme pour n'importe quel flot en faisant appel à toutes les facilités de la classe ostream (dont dérive ofstream).

Par exemple, après la déclaration précédente de sortie, nous pourrons employer des instructions telles que :

```
sortie << .... << .... << .... ;
```

pour réaliser des sorties formatées, ou encore :

```
sortie.write (.....) ;
```

pour réaliser des écritures binaires. De même, nous pourrons connaître le statut d'erreur du flot correspondant en examinant la valeur de sortie :

```
if (sortie) ....
```

Voici un programme complet qui enregistre, sous forme binaire, dans un fichier de nom fourni par l'utilisateur, une suite de nombres entiers qu'il lui fournit sur l'entrée standard.

```
const int LGMAX = 20 ;
#include <stdlib.h>                          // pour exit
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>


main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à créer : " ;
   cin >> setw (LGMAX) >> nomfich ;
```

19 - L'extension peut varier suivant l'implémentation (hxx, h,...).

```
    ofstream sortie (nomfich, ios::out) ;
    if (!sortie) { cout << "création impossible \n" ;
                   exit (1) ;
                 }
    do { cout << "donnez un entier : " ;
         cin >> n ;
         if (n) sortie.write ((char *)&n, sizeof(int) ) ;
       }
    while (n && (sortie)) ;
    sortie.close () ;
}
```

Création séquentielle d'un fichier d'entiers

Nous nous sommes servi du manipulateur setw pour limiter la longueur du nom de fichier fourni par l'utilisateur. Par ailleurs, nous examinons le statut d'erreur de sortie comme nous le ferions pour un flot usuel.

Remarque :

En toute rigueur, le terme "connexion" (ou celui d'association) d'un flot à un fichier pourrait laisser entendre :

- soit qu'il existe deux types d'objets : d'une part un flot, d'autre part un fichier.
- soit que l'on déclare tout d'abord un flot que l'on associe ultérieurement à un fichier.

Or, en fait, il n'en est rien, puisque l'on déclare en une seule fois un objet de ofstream, en spécifiant le fichier correspondant. On pourrait d'ailleurs dire qu'un objet de ce type est un fichier, si l'on ne craignait pas de le confondre avec ce même terme de fichier en langage C (où il désigne souvent un nom interne de fichier, c'est-à-dire un pointeur sur une structure de type FILE).

6.2 Connexion d'un flot d'entrée à un fichier

Pour associer un flot d'entrée à un fichier, on emploie un mécanisme analogue à celui utilisé pour un flot de sortie. On crée, cette fois, un objet de type ifstream, classe dérivant de istream. Il faut toujours inclure le fichier en-tête fstream.h en plus du fichier iostream.h. Le constructeur comporte les mêmes arguments que précédemment, c'est-à-dire nom de fichier et mode d'ouverture.

Par exemple, avec l'instruction :

```
ifstream entree ("truc.dat", ios::in)
```

l'objet entree sera donc associé au fichier de nom truc.dat, après qu'il a été ouvert en lecture.

Une fois construit un objet de classe ifstream, la lecture dans le fichier qui lui est associé pourra se faire comme pour n'importe quel flot d'entrée en faisant appel à toutes les facilités de la classe istream (dont dérive ifstream).

Par exemple, après la déclaration précédente de entree, nous pourrions employer des instructions telles que :

```
entree >> ... >> ... >> ... ;
```

pour réaliser des lectures formatées, ou encore :

```
entree.read (.....) ;
```

pour réaliser des lectures binaires.

Voici un programme complet qui permet de lister le contenu d'un fichier quelconque créé par le programme précédent.

```
const int LGMAX = 20 ;
#include <stdlib.h>                          // pour exit
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à lister : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ifstream entree (nomfich, ios::in) ;
   if (!entree) { cout << "ouverture impossible \n" ;
                  exit (1) ;
                }
   while ( entree.read ( (char*)&n, sizeof(int) ) )
        cout << n << "\n" ;
   entree.close () ;
}
```

Lecture séquentielle d'un fichier d'entiers

Remarque :

En toute rigueur, il existe également une classe fstream, dérivée des deux classes ifstream et ofstream, permettant d'effectuer à la fois des lectures et des écritures avec un même fichier. Cela peut notamment s'avérer fort pratique dans le cas de l'accès direct que nous examinons ci-dessous. La déclaration d'un objet de type fstream se déroule comme pour les types ifstream ou ofstream. Par exemple :

```
fstream fich ("truc.dat", ios::in|ios::out) ;
```

associe l'objet fich au fichier de nom truc.dat, après l'avoir ouvert en lecture et en écriture.

6.3 Les possibilités d'accès direct

Comme en langage C, en C++, dès qu'un flot a été connecté à un fichier, il est possible de réaliser un "accès direct" sur ce fichier en agissant tout simplement sur un pointeur dans ce fichier, c'est-à-dire un nombre précisant le rang du prochain **octet** (caractère) à lire ou à écrire. Après chaque opération de lecture ou d'écriture, ce pointeur est incrémenté du nombre d'octets transférés. Ainsi, lorsque l'on n'agit pas explicitement sur ce pointeur, on réalise un classique accès séquentiel ; c'est ce que nous avons fait précédemment.

Les possibilités d'accès direct se résument donc en fait aux possibilités d'action sur ce pointeur ou à la détermination de sa valeur.

Dans chacune des deux classes ifstream et ofstream, il existe une fonction membre nommée seekg (pour ifstream) et seekp (pour ofstream) permettant de donner une certaine valeur au pointeur (attention, chacune de ces deux classes possède le sien, de sorte qu'il existe un pointeur pour la lecture et un pointeur pour l'écriture). Plus précisément, chacune des ces deux fonctions comporte deux arguments :

- un entier représentant un déplacement du pointeur, par rapport à une origine précisée par le second argument,
- une constante entière choisie parmi trois valeurs prédéfinies dans ios :

  ios::beg : le déplacement est exprimé par rapport au début du fichier,

  ios::cur : le déplacement est exprimé par rapport à la position actuelle,

  ios::end : le déplacement est exprimé par rapport à la fin du fichier (par défaut, cet argument a la valeur ios::beg).

Notez qu'on retrouve là les possibilités offertes par la fonction fseek du langage C.

Par ailleurs, il existe, dans chacune des classes ifstream et ofstream une fonction permettant de connaître la position courante du pointeur. Il s'agit de tellg (pour ifstream) et de tellp (pour ofstream). Celles-ci offrent des possibilités comparables à la fonction ftell du langage C.

Voici un exemple de programme permettant d'accéder à n'importe quel entier d'un fichier du type de ceux que pouvait créer notre programme du paragraphe 6.1.

```
const int LGMAX_NOM_FICH = 20 ;
#include <stdlib.h>               // pour exit
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

main()
{
  char nomfich [LGMAX_NOM_FICH + 1] ;
  int n, num ;
  cout << "nom du fichier à consulter : " ;
  cin >> setw (LGMAX_NOM_FICH) >> nomfich ;
  ifstream entree (nomfich, ios::in) ;
  if (!entree) { cout << "Ouverture impossible\n" ;
                 exit (1) ;
               }

  do
    { cout << "Numéro de l'entier recherché : " ;
      cin >> num ;
      if (num)
         { entree.seekg (sizeof(int) * (num-1) , ios::beg ) ;
           entree.read ( (char *) &n, sizeof(int) ) ;
           if (entree) cout << "-- Valeur : " << n << "\n" ;
             else {  cout << "-- Erreur\n" ;
```

```
                    entree.clear () ;
                  }
            }
       }
    while (num) ;
    entree.close () ;
}
```

```
nom du fichier à consulter : essai.dat
Numéro de l'entier recherché : 4
-- Valeur : 6
Numéro de l'entier recherché : 15
-- Erreur
Numéro de l'entier recherché : 7
-- Valeur : 9
Numéro de l'entier recherché : -3
-- Erreur
Numéro de l'entier recherché : 0
```

Accès direct à un fichier d'entiers

6.4 Les différents modes d'ouverture d'un fichier

Nous avons rencontré quelques exemples de modes d'ouverture d'un fichier. Nous allons examiner ici l'ensemble des possibilités offertes par les classes ifstream et ofstream (et donc aussi de fstream).

Le mode d'ouverture est défini par un mot d'état, dans lequel chaque bit correspond à une signification particulière. La valeur correspondant à chaque bit est définie par des constantes déclarées dans la classe ios. Pour activer plusieurs bits, il suffit de faire appel à l'opérateur |.

BIT	ACTION
ios::in	Ouverture en lecture (obligatoire pour la classe ifstream)
ios::out	Ouverture en écriture (obligatoire pour la classe ofstream)
ios::app	Ouverture en ajout de données (écriture en fin de fichier)
ios::trunc	Si le fichier existe, son contenu est perdu (obligatoire si ios::out est activé sans ios::ate ni ios::app)
ios::binary	(utile dans certaines implémentations uniquement) Le fichier est ouvert en mode dit "binaire" ou encore "non translaté" (voir la remarque ci-après)

Les différents modes d'ouverture d'un fichier

A titre indicatif, voici les modes d'ouverture équivalents aux différents modes d'ouverture de la fonction fopen du C :

Combinaison de bits				Mode correspondant de fopen
ios::out				w
ios::out	ios::app			a
ios::out	ios::trunc			w
ios::in				r
ios::in	ios::out			r+
ios::in	ios::out	ios::trunc		wb+
ios::out	ios::binary			wb
ios::out	ios::app	ios::binary		ab
ios::out	ios::trunc	ios::binary		wb
ios::in	ios::binary			rb
ios::in	ios::out	ios::binary		r+b
ios::in	ios::out	ios::trunc	ios::binary	w+b

Remarque importante : mode texte ou mode binaire

Rappelons que certains environnements (en particulier, les environnements PC) **distinguent les fichiers de texte des autres** (qu'ils appellent parfois "fichiers binaires[20]") ; plus précisément, lors de l'ouverture du fichier, on peut spécifier si l'on souhaite ou non considérer le contenu du fichier comme du texte. Cette distinction est en fait principalement motivée par le fait que le caractère de fin de ligne (\n) possède, sur ces systèmes, une représentation particulière obtenue par la succession de deux caractères (retour chariot \r, suivi de fin de ligne \n). Or, dans ce cas, pour qu'un programme C puisse ne "voir" qu'un seul caractère de fin de ligne et qu'il s'agisse bien de \n, il faut opérer un traitement particulier consistant à :

- remplacer chaque occurrence de ce couple de caractères par \n, dans le cas d'une lecture,
- remplacer chaque demande d'écriture de \n par l'écriture de ce couple de caractères.

Bien entendu, de telles substitutions ne doivent pas être réalisées sur des "vrais fichiers binaires". Il faut donc bien pouvoir opérer une distinction au sein du programme. Cette distinction se fait au moment de l'ouverture du fichier, en activant le bit ios::binary dans le mode d'ouverture, dans le cas d'un fichier binaire ; par défaut, ce bit n'est pas activé. On notera que cela correspond aux modes d'ouverture "rb" ou "wb" du langage C.

7. LES POSSIBILITÉS DE FORMATAGE EN MÉMOIRE

En langage C :

20 - Alors qu'au bout du compte tout fichier est binaire !

- sscanf permet d'accéder à une information située en mémoire, de façon comparable à ce que fait scanf sur l'entrée standard,
- sprintf permet de fabriquer en mémoire une chaîne de caractères correspondant à celle qui serait transmise à la sortie standard par printf.

En C++, des facilités comparables existent. Elles sont fournies par les classes :

- ostrstream pour "l'insertion" de caractères dans un tableau,
- istrstream pour "l'extraction" de caractères depuis un tableau.

Leur utilisation nécessite l'inclusion du fichier en-tête istrstream.h[21].

7.1 La classe ostrstream

Un objet de classe ostrstream peut recevoir des caractères, au même titre qu'un flot de sortie. La seule différence est que ces caractères ne sont pas transmis à un périphérique ou à un fichier, mais simplement conservés dans l'objet lui-même, plus précisément dans un tableau membre de la classe ostrstream; ce tableau est créé dynamiquement et ne pose donc pas de problème de limitation de taille.

Une fonction membre particulière nommée str permet d'obtenir l'adresse du tableau en question. Celui-ci pourra alors être manipulé comme n'importe quel tableau de caractères (repéré par un pointeur de type char *).

Par exemple, avec la déclaration :

```
ostrstream tab
```

vous pouvez insérer des caractères dans l'objet tab par des instructions telles que :

```
tab << ..... << ..... << ..... ;
```

ou :

```
tab.put (.....) ;
```

ou encore :

```
tab.write (.....) ;
```

L'adresse du tableau de caractères ainsi constitué pourra être obtenue par :

```
char * adt = tab.str () ;
```

A partir de là, vous pourrez agir comme il vous plaira sur les caractères situés à cette adresse (les consulter, mais éventuellement les modifier...).

Remarques très importantes

1) Lorsque str a été appelée pour un objet, il n'est plus possible d'insérer de nouveaux caractères dans cet objet. On peut dire que l'appel de cette fonction "fige" définitivement le tableau de caractères (n'oubliez pas qu'il est alloué dynamiquement et que son adresse peut même évoluer au fil de l'insertion de caractères!), avant d'en fournir en retour une adresse définitive. On prendra donc bien soin de n'appeler str que lorsque l'on aura inséré dans l'objet tous les caractères voulus.

[21] - L'extension peut varier suivant l'implémentation.

2) Si un objet de classe ostrstream devient hors de portée, alors que la fonction str n'a pas été appelée, il est détruit normalement par appel d'un destructeur qui détruit alors également le tableau de caractères correspondant. En revanche, si str a été appelée, on considère que le tableau en question est maintenant sous la responsabilité du programmeur et il ne sera donc pas détruit lorsque l'objet deviendra hors de portée (bien sûr, le reste de l'objet le sera). Ce sera au programmeur de le faire lorsqu'il le souhaitera, en procédant comme pour n'importe quel tableau de caractères alloué dynamiquement (par new) c'est-à-dire en faisant appel à l'opérateur delete. Par exemple, l'emplacement mémoire du tableau de l'objet tab précédent, dont l'adresse a été obtenue dans adt pourra être libéré par :

```
delete adt ;
```

7.2 La classe istrstream

Un objet de classe istrstream est créé par un appel de constructeur, auquel on fournit en argument :

- l'adresse d'un tableau de caractères,
- le nombre de caractères à prendre en compte.

Il est alors possible d'extraire des caractères de cet objet, comme on le ferait de n'importe quel flot d'entrée.

Par exemple, avec les déclarations :

```
char t[100] ;
istrstream tab ( t, sizeof(t) ) ;
```

vous pourrez extraire des caractères du tableau t par des instructions telles que :

```
tab >> ..... >> ..... >> ..... ;
```

ou :

```
tab.get (.....) ;
```

ou encore :

```
tab.read (.....) ;
```

Qui plus est, vous pourrez agir sur un pointeur courant dans ce tableau, comme vous le feriez dans un fichier par l'appel de la fonction seekg. Par exemple, avec l'objet tab précédent, vous pourrez replacer le pointeur en début de tableau par :

```
tab.seekg (0, ios::beg) ;
```

Cela pourrait permettre, par exemple, d'exploiter plusieurs fois une même information (lue préalablement dans un tableau) en la "lisant" suivant des formats différents.

Voici un exemple d'utilisation de la classe istrstream montrant comment résoudre les problèmes engendrés par la frappe d'un "mauvais" caractère dans le cas de lectures sur l'entrée standard.

```
const LGMAX = 122 ;          // longueur maxi d'une ligne clavier
#include <iostream.h>
#include <strstream.h>
```

```
main()
{  int n, erreur ;
   char c ;
   char ligne [LGMAX] ;      // pour lire une ligne au clavier
   do
     { cout << "donnez un entier et un caractère :\n" ;
       cin.getline (ligne, LGMAX) ;
       istrstream tampon (ligne, cin.gcount () ) ;
       if (tampon >> n >> c) erreur = 0 ;
                        else erreur = 1 ;
     }
   while (erreur) ;
   cout << "merci pour " << n << " et " << c << "\n" ;
}
```

```
donnez un entier et un caractère :
bof
donnez un entier et un caractère :
a 125
donnez un entier et un caractère :
12 bonjour
merci pour 12 et b
```

Pour lire en toute sécurité sur l'entrée standard

Nous y lisons tout d'abord l'information attendue pour toute une ligne, sous forme d'une chaîne de caractères (à l'aide de la fonction getline). Nous construisons ensuite, avec cette chaîne, un objet de type istrstream sur lequel nous appliquons nos opérations de lecture (ici lecture formatée d'un entier puis d'un caractère). Comme vous le constatez, aucun problème ne se pose plus lorsque l'utilisateur fournit un caractère invalide (par rapport à l'usage qu'on doit en faire), contrairement à ce qui se serait passé en cas de lecture directe sur cin.

XVII. LA GESTION DES EXCEPTIONS

Dans sa version définitive[1], C++ comporte un mécanisme dit de "gestion des exceptions" que nous proposons d'étudier dans ce chapitre. On nomme exception, une rupture de séquence décidée par programme (par l'intermédiaire du mot clé throw) ; il y a alors débranchement à une fonction nommée gestionnaire d'interruption et dont le nom est déterminé par la nature de l'interruption. Plus précisément, chaque interruption est caractérisée par un type classe et le choix du "bon gestionnaire" se fait en fonction de la nature de l'objet mentionné à throw.

1. UN PREMIER EXEMPLE D'EXCEPTION

Dans l'exemple complet ci-après, nous allons reprendre la classe vect présentée dans le chapitre "surdéfinition des opérateurs", c'est-à-dire munie de la surdéfinition de l'opérateur []. Celui-ci n'était alors pas "protégé" contre l'utilisation d'indices situés en dehors des bornes ; ici, nous allons compléter notre classe pour qu'elle lance une exception dans ce cas. Puis nous verrons comment intercepter une telle exception en écrivant un gestionnaire d'exception approprié.

1.1 Comment lancer une exception : l'instruction throw

Nous introduisons donc, au sein de la surdéfinition de [], une vérification de l'indice ; lorsque celui-ci est incorrect, nous "lançons[2]" une exception, à l'aide de l'instruction throw. De par sa nature même, cette dernière nécessite une expression de type classe (et dont le type sert à identifier l'exception) ; c'est la raison pour laquelle nous avons introduit artificiellement, avec la déclaration de notre classe vect, une classe (sans aucun membre ici) nommée vect_limite. Son existence nous permet de créer un objet l, de type vect_limite, objet que nous associons à l'instruction throw par l'instruction : `throw l ;`

Voici la définition complète de notre classe vect :

```
 /* déclaration de la classe vect */
class vect
```

1 - Plus précisément, ce mécanisme a été introduit dans la version 3.

2 - On emploie également le terme "lever".

```
{ int nelem ;
  int * adr ;
 public :
  vect (int) ;
  ~vect () ;
  int & operator [] (int) ;
} ;
 /* déclaration et définition d'une classe vect_limite (vide pour l'instant) */
class vect_limite
{  } ;
 /* définition de la classe vect */
vect::vect (int n)
     { adr = new int [nelem = n] ;
     }
vect::~vect ()
     { delete adr ;
     }
int & vect::operator [] (int i)
     { if (i<0 || i>nelem)
        { vect_limite l ; throw (l) ;
        }
       return adr [i] ;
     }
```

Définition d'une classe provoquant une exception vect_limite

1.2 Utilisation d'un gestionnaire d'exception

Disposant de notre classe vect, voyons maintenant comment procéder pour pouvoir gérer convenablement les éventuelles exceptions de type vect_limite que son emploi peut provoquer. Pour ce faire, il est nécessaire de respecter deux conditions :

- inclure dans un bloc particulier, dit "bloc try", toutes les instructions dans lesquelles on souhaite pouvoir lancer une exception ; un tel bloc se présente ainsi :

```
try
{
   //  instructions
}
```

- faire suivre ce bloc de la définition des différents "gestionnaires d'exceptions" nécessaires (ici, un seul nous suffit). Chaque définition est précédée d'un en-tête introduit par le mot clé catch (comme si catch était le nom d'une fonction gestionnaire...). Dans notre cas, voici ce que pourrait être notre unique gestionnaire, destiné à intercepter les exceptions de type vect_limite :

```
catch (vect_limite l)  /* nom d'argument superflu ici */
 { cout << "exception limite \n" ;
      exit (-1) ;
 }
```

Nous nous contentons ici d'afficher un message et d'interrompre l'exécution du programme.

1.3 Récapitulatif

A titre indicatif, nous vous fournissons ici la liste complète de la définition des différentes classes conernées et d'un petit programme d'essai dans lequel nous provoquons volontairement une exception vect_limite (par application de l'opérateur [] à un objet de type vect, en lui indiquant un indice trop grand).

```
#include <iostream.h>
#include <stdlib.h>       /* pour exit */
 /* déclaration de la classe vect */
class vect
{ int nelem ;
  int * adr ;
 public :
  vect (int) ;
  ~vect () ;
  int & operator [] (int) ;
} ;
 /* déclaration et définition d'une classe vect_limite (vide pour l'instant) */
class vect_limite
 { } ;
 /* définition de la classe vect */
vect::vect (int n)
   { adr = new int [nelem = n] ; }
vect::~vect ()
   { delete adr ; }
int & vect::operator [] (int i)
   { if (i<0 || i>nelem)
       { vect_limite l ; throw (l) ;
       }
     return adr [i] ;
   }
 /* test interception exception vect_limite */
main ()
{ try
     { vect v(10) ;
       v[11] = 5 ;    /* indice trop grand */
     }
  catch (vect_limite l)  /* nom d'argument superflu ici */
     { cout << "exception limite \n" ;
       exit (-1) ;
     }
 }

exception limite
```

1.4 Commentaires

a) Ce premier exemple, destiné à vous présenter le mécanisme de gestion des exceptions, était relativement simpliste ; notamment :

- il ne comporte qu'un seul type d'exception, de sorte qu'il ne met pas vraiment en évidence le mécanisme de choix du bon gestionnaire,
- le gestionnaire ne reçoit pas d'information particulière (l'argument l étant ici artificiel).

Nous y reviendrons précisément dans le prochain paragraphe.

b) D'une manière générale, le gestionnaire d'une exception est défini de façon indépendante des fonctions qui sont susceptibles de lever cette exception. Ainsi, à partir du moment où la définition d'une classe est séparée de son utilisation (ce qui est souvent le cas en pratique), il est tout à fait possible de prévoir un gestionnaire d'exception différent d'une utilisation à une autre d'une même classe. Dans notre précédent exemple, tel utilisateur pourra vouloir afficher un message avant de s'interrompre, tel autre préférera ne rien afficher ou encore tenter de prévoir une solution par défaut...

c) Ici, nous avons prévu une instruction exit, à l'intérieur de notre gestionnaire d'exception. Nous verrons plus loin ce qui se produirait si tel n'était pas le cas.

2. UN SECOND EXEMPLE

Voyons maintenant un exemple un peu plus réaliste dans lequel on trouve à la fois deux exceptions différentes et où il y a transmission d'information aux gestionnaires. Nous allons reprendre notre classe vect précédente, en lui permettant de lancer deux sortes d'exceptions :

- une exception de type vect_limite comme précédemment, mais, cette fois, on prévoit de transmettre au gestionnaire la valeur de l'indice qui a provoqué l'exception,
- une exception vect_creation lancée lorsque l'on transmet au constructeur un nombre d'éléments incorrect[3] (négatif ou nul) ; là encore, on prévoit de transmettre ce nombre au gestionnaire.

Il nous suffit d'appliquer le mécanisme précédent, en notant simplement que l'objet indiqué à throw et récupéré par catch peut nous servir à communiquer toute information de notre choix. Ici, nous prévoirons donc, dans nos nouvelles classes vect_limite et vect_creation, un champ public de type entier destiné à recevoir l'information à transmettre au gestionnaire.

Voici un exemple complet (ici, encore, la définition et l'utilisation des classes figurent dans le même source ; en pratique, il en ira rarement ainsi) :

```
#include <iostream.h>
#include <stdlib.h>       // pour exit
 /* déclaration de la classe vect */
class vect
{ int nelem ;
  int * adr ;
```

[3] - Dans un cas réel, on pourrait également lancer cette même interrruption en cas de manque de mémoire.

```
 public :
  vect (int) ;
  ~vect () ;
  int & operator [] (int) ;
} ;

 /* déclaration - définition des deux classes exception */
class vect_limite
{ public :
   int hors ;             // valeur indice hors limites (public)
   vect_limite (int i)    // constructeur
    { hors = i ; }
} ;
class vect_creation
{ public :
   int nb ;               // nombre elements demandes (public)
   vect_creation (int i)  // constructeur
    { nb = i ; }
} ;

 /* définition de la classe vect */
vect::vect (int n)
    { if (n <= 0)
       { vect_creation c(n) ;     // anomalie
         throw (c) ;
       }
      adr = new int [nelem = n] ; // construction normale
    }
vect::~vect ()
    { delete adr ;
    }

int & vect::operator [] (int i)
    { if (i<0 || i>nelem)
         { vect_limite l(i) ;     // anomalie
           throw (l) ;
         }
      return adr [i] ;            // fonctionnement normal
    }

 /* test exception */
main ()
{
  try
     { vect v(-3) ;         // provoque l'exception vect_creation
       v[11] = 5 ;     // provoquerait l'exception vect_limite
     }
   catch (vect_limite l)
    { cout << "exception indice " << l.hors << " hors limites \n" ;
      exit (-1) ;
    }
   catch (vect_creation c)
    { cout << "exception création vect nb elem = " << c.nb << "\n" ;
      exit (-1) ;
    }
```

```
}
```

```
exception création vect nb elem = -3
```

Exemple de gestion de deux exceptions, avec transmission d'information au gestionnaire

Bien entendu, la première exception (déclenchée par vect v(-3)) ayant provoqué l'arrêt de l'exécution, nous n'avons ici aucune chance de mettre en évidence celle qu'aurait provoqué v[11] = 5 ; si la création de v avait été correcte, cette dernière instruction aurait entraîné l'affichage du message :

```
exception indice 11 hors limites
```

Remarques :

1) Dans un exemple réel, on pourrait avoir intérêt à transmettre, dans vect_limite, non seulement la valeur de l'indice, mais également les limites prévues ; il suffirait d'introduire les membres correspondants dans la classe vect_limite.
2) Ici, nos classes vect_limite et vect_creation sont indépendantes de la classe vect. Mais il serait tout à fait possible d'en faire des classes membre de la classe vect.
3) Ici, chaque type d'exception n'est lancé qu'en un seul endroit ; mais, bien entendu, n'importe quelle fonction (pas nécessairement membre de la classe vect !) disposant de la définition des deux classes (vect_limite et vect_creation) peut lancer ces exceptions.
4) De par sa nature même, le mécanisme de gestion des exceptions, tel qu'il a été prévu par C++, fait obligatoirement appel à un objet de type classe (ou, comme nous le verrons un peu plus loin, à un pointeur sur un objet de type classe).

3. LES EXCEPTIONS D'UNE MANIÈRE GÉNÉRALE

3.1 Après l'exécution du gestionnaire d'exception

Dans tous nos exemples précédents, le gestionnaire d'interruption interrompait l'exécution par un appel de exit (nous aurions pu également utiliser la fonction standard abort). Si tel n'avait pas été le cas, il faut savoir qu'alors, après l'exécution des intructions du gestionnaire concerné, on exécute, si elles existent, les instructions (du bloc try) suivant celles du dernier gestionnaire. Si aucune instruction d'arrêt d'exécution ne figure parmi ces dernières, on se trouve dans la situation classique de retour de la fonction correspondante.

Voyez cet exemple (il utilise les mêmes classes vect, vect_limite et vect_creation que précédemment) dans lequel nous avons artificiellement appelé, depuis main, une fonction f, dans laquelle on accepte de gérer les exceptions (afin de montrer qu'après exécution d'un gestionnaire d'exception il peut y avoir retour dans la fonction appelant f, à savoir ici main) :

```
// déclaration et définition des classes vect, vect_limite, vect_creation
```

```
//    comme dans le paragraphe 2
//    .....
main()
{ void f() ;
  f() ;
  cout << "après appel de f \n" ;
}
void f()
{ try
     { vect v(-3) ;                // nombre éléments incorrect
            // ...
     }
  catch (vect_limite l)
      { cout << "exception indice " << l.hors << " hors limites \n" ; }
  catch (vect_creation c)
      { cout << "exception création vect nb elem = " << c.nb << "\n" ;
      }
     // traitement commun aux exceptions n'ayant pas appelé exit
     cout << "traitement commun \n" ;
 }
```

```
exception création vect nb elem = -3
traitement commun
après appel de f
```

Lorsque l'on "passe à travers" un gestionnaire d'exception

3.2 Algorithme de choix du gestionnaire d'interruption

Le choix du "bon gestionnaire" se base sur le type de l'expression mentionné dans l'instruction throw. Plus précisément, on cherche un gestionnaire correspondant à l'un des types suivants :

- type exact mentionné dans throw,
- type correspondant à une classe de base du type mentionné dans throw ; cette possibilité est précieuse pour regrouper plusieurs exceptions qu'on peut traiter plus ou moins "finement"...,
- type correspondant à un pointeur sur une classe dérivée du type mentionné dans throw (lorsque ce type est lui-même un pointeur),
- le type mentionné dans catch correspond à un type indéterminé, noté par des points de suspension.

Dès qu'un gestionnaire correspond, on l'exécute, sans se préocuper de l'existence d'autres gestionnaires. Ainsi, avec :

```
catch (truc)    // gestionnaire 1
   { // }
catch (...)     // gestionnaire 2
   { // }
catch (chose)   // gestionnaire 3
   { // }
```

le gestionnaire 3 n'a aucune chance d'être exécuté, puisque le gestionnaire 2 interceptera toutes les exceptions non interceptées par le gestionnaire 1.

3.3 Le cheminement des exceptions

Quand une exception est levée par une fonction, on cherche tout d'abord un gestionnaire dans l'éventuel bloc try associé à cette fonction ; si l'on n'en trouve pas (ou si aucun bloc try n'est associé), on poursuit la recherche dans un éventuel bloc try associé à une fonction appelante et ainsi de suite. Voyez cet exemple (utilisant toujours les mêmes classes que précédemment) :

```
 /* test exception */
main ()
{  try
    { void f1 () ;
      f1 () ;
    }
    catch (vect_limite l)
    { cout << "dans main : exception indice \n" ;
      exit (-1) ;
    }
}
void f1 ()
{
  try
   { vect v(10) ; v[12] = 0 ; // provoque affichage de : dans main : exception
ndice
     vect v1 (-1) ;           // provoque affichage de : dans f1 : exception
création
                              //   (à condition que l'instruction précédente
n'ait
                              //         pas déjà provoqué une exception)
   }
  catch (vect_creation v)
   { cout << "dans f1 : exception création \n" ;
   }
}
```

Si aucun gestionnaire d'exception n'est trouvé, on appelle la fonction **terminate**. Par défaut, cette dernière appelle la fonction abort (avec Turbo/Borland C++, elle fournit le message "Abnormal program termination"). Il est possible de demander que soit appelée une fonction de votre choix dont vous fournissez l'adresse à set_terminate (de façon comparable à ce que vous faites avec set_new_handler).

3.4 Spécification d'interface

Une fonction (y compris main) peut spécifier les exceptions qu'elle est susceptible de provoquer (elle-même, ou dans les fonctions qu'elle appelle à son tour). Elle le fait à l'aide du mot clé throw, suivi, entre parenthèses, de la liste des exceptions concernées. Dans ce cas, toute exception non prévue et levée à l'intérieur de la fonction (ou d'une fonction appelée) entraîne l'appel d'une fonction particulière nommée unexpected. Par défaut, cette fonction appelle la fonction terminate (ou la fonction indiquée à set_terminate). Mais vous pouvez également fournir votre propre fonction, en remplacement de unexpected, en l'indiquant par set_unexpected.

Voici un exemple qui utilise, ici encore, les classes vect, vect_limite et vect_creation définies dans le paragraphe 3 :

```
#include <iostream.h>
#include <stdlib.h>       // pour exit
#include <except.h>       // pour set_unexpected
// définition et déclaration des classes vect, vect_limite, vect_creation
//    comme dans le paragraphe 3
//    .....

 /* test exception */
main () throw (vect_limite)
{ try
    { void autre_except () ;  // pour gérer les exception non attendues
      set_unexpected (autre_except) ;
      vect v(-3) ;                // nombre éléments incorrect
      v[11] = 5 ;
    }
   catch (vect_limite l)
    { cout << "exception indice " << l.hors << " hors limites \n" ;
      exit (-1) ;
    }
}
void autre_except ()
  {  cout << "exception non attendue \n" ; }
```

Son exécution fournit tout d'abord le message exception non attendue, puis Abnormal termination.

Notez que nous obtiendrions exactement la même chose avec ce canevas (les classes vect, vect_limite et vect_creation étant toujours les mêmes) :

```
main ()
{ try
    { void f1 () throw (vect_limite) ;
      void autre_except () ;  // pour gérer les exception non attendues
      set_unexpected (autre_except) ;
      f1 () ;
    }
    catch (vect_limite l)
    { cout << "exception indice " << l.hors << " hors limites \n" ;
      exit (-1) ;
    }
}
void f1 () throw (vect_limite)
{ vect v1 (-1) ;
}
void autre_except ()
{  cout << "exception non attendue \n" ;
}
```

4. LES EXCEPTIONS STANDARD

La bibliothèque standard comporte quelques classes correspondants à des exceptions spécifiques qui peuvent éventuellement être utilisées dans un programme. Leurs déclarations figurent, notamment, dans stdexcept.

La plupart ont une signification évidente : domain_error, invalid_argument, length_error, out_of_range, range_error, overflow_error, underflow_error, bad_alloc. Leurs constructeurs disposent d'un argument de type string correspondant au message à afficher. Le type string ne sera étudié que dans le chapitre XXI mais sachez, dès maintenant, qu'une chaîne usuelle peut très bien faire l'affaire.

Par ailleurs, on y trouve également les exceptions bad_type_id et bad_cast dont la signification sera plus compréhensible après examen de l'annexe C.

XVIII. GÉNÉRALITÉS CONCERNANT LA BIBLIOTHÈQUE STANDARD

Comme celle du C, la norme du C++ comprend la définition d'une bibliothèque standard. Bien entendu, on y trouve toutes les fonctions prévues dans les versions C++ d'avant la norme, qu'il s'agisse des flots décrits précédemment ou des fonctions de la bibliothèque standard du C. Mais, on y découvre surtout bon nombre de nouveautés originales. La plupart d'entre elles sont constituées de patrons de classes et de fonctions provenant en majorité d'une bibliothèque du domaine public, nommée Standard Template Library (en abrégé STL) et développée chez Hewlett Packard.

L'objectif de ce chapitre est de vous familiariser avec les notions de base concernant l'utilisation des principaux composants de cette bibliothèque, à savoir : les conteneurs, les itérateurs, les algorithmes, les générateurs d'opérateurs, les prédicats et l'utilisation d'une relation d'ordre.

1. NOTIONS DE CONTENEUR, D'ITERATEUR ET D'ALGORITHME

Ces trois notions sont étroitement liées et, la plupart du temps, elles interviennent simultanément dans un programme utilisant des conteneurs.

1.1 Notion de conteneur

La bibliothèque standard fournit un ensemble de classes dites conteneurs, permettant de représenter les structures de données les plus répandues telles que les vecteurs, les listes, les ensembles ou les tableaux associatifs. Il s'agit de patrons de classes paramétrés tout naturellement par le type de leurs éléments. Par exemple, on pourra construire une liste d'entiers, un vecteur de flottants ou une liste de points (point étant une classe) par les déclarations suivantes :

```
list <int>      li ;     /* liste vide dont les éléments seront de type int
*/
vector <double> ld ;     /* vecteur vide dont les éléments seront de type
double */
list <point>    lp ;     /* liste vide dont les éléments seront de type point
*/
```

Chacune de ces classes conteneur dispose de fonctionnalités appropriées dont on pourrait penser, a priori, qu'elles sont très différentes d'un conteneur à l'autre. En réalité, les concepteurs de STL ont fait un gros effort d'homogénéisation et beaucoup de fonctions membre sont communes à différents conteneurs. On peut dire que, dès qu'une action donnée est réalisable avec deux conteneurs différents, elle se programme de la même manière.

Remarque importante

En toute rigueur, les patrons de conteneurs sont paramétrés à la fois par le type de leurs éléments et par une fonction dite allocateur utilisée pour les allocations et les libérations de mémoire. Ce second paramètre possède une valeur par défaut qui est généralement satisfaisante. Cependant, certaines implémentations n'acceptent pas encore les paramètres par défaut dans les patrons de classes et, dans ce cas, il est nécessaire de préciser l'allocateur à utiliser, même s'il s'agit de celui par défaut. Il faut alors savoir que ce dernier est une fonction patron, de nom allocator, paramétrée par le type des éléments concernés. Voici ce que deviendraient les déclarations précédentes dans un tel cas :

```
list <int, allocator<int> > li ;     /* attention à laisser un espace entre >
et > */
vector <double, allocator<double> > ld ;   /* sinon , >> serait interprété
comme   */
list <point, allocator<point> > lp ;       /* l'opérateur >>
*/
```

1.2 Notion d'itérateur

C'est dans ce souci d'homogénéisation des actions sur un conteneur qu'a été introduite la notion d'itérateur. Un itérateur est un objet défini généralement par la classe conteneur concernée qui généralise la notion de pointeur :

- à un instant donné, un itérateur possède une valeur qui désigne un élément donné d'un conteneur ; on dira souvent qu'un itérateur pointe sur un élément d'un conteneur ;
- un itérateur peut être incrémenté par l'opérateur ++, de manière à pointer sur l'élément suivant du même conteneur ; on notera que ceci n'est possible que, comme on le verra plus loin, parce que les conteneurs sont toujours ordonnés suivant une certaine séquence ;
- un itérateur peut être déréférencé, comme un pointeur, en utilisant l'opérateur * ; par exemple, si it est un itérateur sur une liste de points, *it désigne un point de cette liste ;
- deux itérateurs sur un même conteneur peuvent être comparés par égalité ou inégalité.

Tous les conteneurs fournissent un itérateur portant le nom iterator et possédant au minimum les propriétés que nous venons d'énumérer qui correspondent à ce qu'on nomme un itérateur unidirectionnel. Certains itérateurs pourront posséder des propriétés supplémentaires, en particulier :

- décrémentation par l'opérateur -- ; comme cette possibilité s'ajoute alors à celle qui est offerte par ++, l'itérateur est alors dit bidirectionnel ;
- accès direct ; dans ce cas, si it est un tel itérateur, l'expression it+i a un sens ; souvent, l'opérateur [] est alors défini, de manière que it[i] soit équivalent à *(it+i) ; en outre, un tel itérateur peut être comparé par inégalité.

Remarque :

Ici, nous avons évoqué trois catégories d'itérateur : unidirectionnel, bidirectionnel et accès direct. Dans le chapitre XXI, nous verrons qu'il existe deux autres catégories (entrée et sortie) qui sont d'un usage plus limité. De même, on verra qu'il existe ce qu'on appelle des adaptateurs d'itérateur, lesquels permettent d'en modifier les propriétés ; les plus importants seront l'itérateur de flux et l'itérateur d'insertion.

1.3 Parcours d'un conteneur avec un itérateur

a) Parcours direct

Tous les conteneurs fournissent des valeurs particulières de type iterator, sous forme des fonctions membre begin() et end(), de sorte que, quel que soit le conteneur, le canevas suivant, présenté ici sur une liste de points, est toujours utilisable pour parcourir séquentiellement un conteneur de son début jusqu'à sa fin :

```
list<point> lp ;
   .....
list<point>::iterator il ;     /* itérateur sur une liste de points */
for (il = lp.begin() ; il != lp.end() ; il++)
{
   /* ici *il désigne l'élément courant de la liste de points lp */
}
```

On notera la particularité des valeurs des itérateurs de fin qui consiste à pointer, non pas sur le dernier élément d'un conteneur, mais juste après. D'ailleurs, lorsqu'un conteneur est vide, begin() possède la même valeur que end(), de sorte que le canevas précédent fonctionne toujours convenablement.

Remarque

Attention, on ne peut pas utiliser comme condition d'arrêt de la boucle for, une expression telle que il < lp.end, car l'opérateur < ne peut s'appliquer qu'à des itérateurs à accès direct.

b) Parcours inverse

Toutes les classes conteneur pour lesquelles iterator est au moins bidirectionnel (on peut donc lui appliquer ++ et --) disposent d'un second itérateur noté reverse_iterator. Construit à partir du premier, il permet d'explorer le conteneur suivant l'ordre inverse. Dans ce cas, la signification de ++ et --, appliqués à cet itérateur, est alors adaptée en conséquence ; en outre, il existe également des valeurs particulières de type reverse_iterator fournies par les fonctions membre rbegin() et rend() ; on peut dire que rbegin() pointe sur le dernier élément du conteneur, tandis que rend() pointe juste avant le premier. Voici comment parcourir une liste de points dans l'ordre inverse :

```
list<point> lp ;
   .....
list<point>::reverse_iterator ril ;   /* itérateur inverse sur une liste de
points */
for (ril = lp.rbegin() ; ril != lp.rend() ; ril++)
{
   /* ici *ril désigne l'élément courant de la liste de points lp */
}
```

1.4 Intervalle d'itérateur

Comme nous l'avons déjà fait remarquer, tous les conteneurs sont ordonnés, de sorte qu'on peut toujours les parcourir d'un début jusqu'à une fin. Plus généralement, on peut définir ce qu'on nomme un intervalle d'itérateur en précisant les bornes sous forme de deux valeurs d'itérateur. Supposons que l'on ait déclaré :

```
vector<point>::iterator ip1, ip2 ;   /* ip1 et ip2 sont des itérateurs sur un
*/
                                     /*   vecteur de points
*/
```

Supposons, de plus, que ip1 et ip2 possèdent des valeurs telles que ip2 soit "accessible" depuis ip1, autrement dit que, après un certain nombre d'incrémentations de ip1 par ++, on obtienne la valeur de ip2. Dans ces conditions, le couple de valeurs ip1, ip2 définit un intervalle d'un conteneur du type vector< point> s'étendant de l'élément pointé par ip1 jusqu'à (mais non compris) celui pointé par ip2. Cet intervalle se note souvent [ip1, ip2). On dit également que les éléments désignés par cet intervalle forment une séquence.

Cette notion d'intervalle d'itérateur sera très utilisée par les algorithmes et par certaines fonctions membre.

1.5 Notion d'algorithme

La notion d'algorithme est tout aussi originale que les deux précédentes. Elle se fonde sur le fait que, par le biais d'un itérateur, beaucoup d'opérations peuvent être appliquées à un conteneur, quels que soient sa nature et le type de ses éléments. Par exemple, on pourra trouver le premier élément ayant une valeur donnée aussi bien dans une liste, un vecteur ou ensemble ; il faudra cependant que l'égalité de deux éléments soit convenablement définie, soit par défaut, soit par surdéfinition de l'opérateur ==. De même, on pourra trier un conteneur d'objets de type T, pour peu que ce conteneur dispose d'un itérateur à accès direct et que l'on ait défini une relation d'ordre sur le type T, par exemple en surdéfinissant l'opérateur <.

Les différents algorithmes sont fournis sous forme de patrons de fonctions, paramétrés par le type des itérateurs qui leurs sont fournis en argument. Là encore, cela conduit à des programmes très homogènes puisque les mêmes fonctions pourront être appliquées à des conteneurs différents. Par exemple, pour compter le nombre d'éléments égaux à un dans un vecteur déclaré par :

```
vector<int> v ;    /* vecteur d'entiers */
```

on pourra procéder ainsi :

```
n = count (v.begin(), v.end(), 1) ;    /* compte le nombre d'éléments de valeur
1 */
                                       /* dans la séquence [v.begin(), v.end())
*/
                                       /* autrement dit, dans tout le conteneur
v */
```

Pour compter le nombre d'éléments égaux à un dans une liste déclarée :

```
list<int> l ;    /* liste d'entiers */
```

on procédera de façon similaire (en se contentant de remplacer v par l) :

```
n = count (l.begin(), l.end(), 1) ;    /* compte le nombre d'éléments de valeur
1 */
```

```
                                                    /* dans la séquence [l.begin(), l.end())
*/
                                                    /* autrement dit, dans tout le conteneur
l */
```

D'une manière générale, comme le laissent entendre ces deux exemples, les algorithmes s'appliquent, non pas à un conteneur, mais à une séquence définie par un intervalle d'itérateur ; ici, cette séquence correspondait à l'intégralité du conteneur.

Certains algorithmes permettront facilement de recopier des informations d'un conteneur d'un type donné vers un conteneur d'un autre type, pour peu que ses éléments soient du même type que ceux du premier conteneur. Voici, par exemple, comment recopier un vecteur d'entiers dans une liste d'entiers :

```
vector<int> v ;     /* vecteur d'entiers */
list<int> l ;       /* liste d'entiers   */
   .....
copy (v.begin(), v.end(), l.begin() ) ; /* recopie l'inter [v.begin(), v.end())
*/
                                         /* à partir de l'emplacement pointé par
*/
                                         /* l.begin()
*/
```

Notez que, si l'on fournit l'intervalle de départ, on ne mentionne que le début de celui d'arrivée.

Remarque

On pourra parfois être gêné par le fait que l'homogénéisation évoquée n'est pas absolue. Ainsi, on verra qu'il existe un algorithme de recherche d'une valeur donnée nommé find, alors même qu'un conteneur comme list dispose d'une fonction membre comparable. La justification résidera dans des considérations d'efficacité.

1.6 Itérateurs et pointeurs

La manière dont les algorithmes ou les fonctions membre utilisent un itérateur fait que tout objet ou toute variable possédant les propriétés attendues (déréférenciation, incrémentation...) peut être utilisé à la place d'un objet tel que iterator.

Or, les pointeurs usuels possèdent tout naturellement les propriétés d'un itérateur à accès direct. Cela leur permet d'être employés dans bon nombre d'algorithmes. Cette possibilité est fréquemment utilisée pour la recopie des éléments d'un tableau ordinaire dans un conteneur :

```
int t[6] = { 2, 9, 1, 8, 2, 11 } ;
list<int> l ;
   .....
copy (t, t+6, l.begin()) ;      /* copie de l'intervalle [t, t+6) dans la liste
l */
```

Bien entendu, ici, il n'est pas question d'utiliser une notation telle que t.begin() qui n'aurait aucun sens, t n'étant pas un objet.

Remarque

Par souci de simplicité, nous parlerons encore de séquence d'éléments (mais plus de séquence de conteneur) pour désigner les éléments ainsi définis par un intervalle de pointeurs.

2. LES DIFFERENTES SORTES DE CONTENEURS

2.1 Conteneurs et structures de données classiques

On dit souvent que les conteneurs correspondent à des structures de données usuelles. Mais, à partir du moment où ces conteneurs sont des classes qui encapsulent convenablement leurs données, leurs caractéristiques doivent être indépendantes de leur implémentation. Dans ces conditions, les différents conteneurs devraient se distinguer les uns des autres uniquement par leurs fonctionnalités et en aucun cas par les structures de données sous-jacentes. Beaucoup de conteneurs posséderaient alors des fonctionnalités voisines, voire identiques.

En réalité, les différents conteneurs se caractérisent, outre leurs fonctionnalités, par l'efficacité de certaines opérations. Par exemple, on verra qu'un vecteur permet des insertions d'éléments en n'importe quel point mais celles-ci sont moins efficaces qu'avec une liste. En revanche, on peut accéder plus rapidement à un élément existant dans le cas d'un vecteur que dans celui d'une liste. Ainsi, bien que la norme n'impose pas l'implémentation des conteneurs, elle introduit des contraintes d'efficacité qui la conditionneront largement.

En définitive, on peut dire que le nom choisi pour un conteneur évoque la structure de donnée classique qui en est proche sur le plan des fonctionnalités, sans pour autant coïncider avec elle. Dans ces conditions, un bon usage des différents conteneurs passe par un apprentissage de leurs possibilités, comme s'il s'agissait bel et bien de classes différentes.

2.2 Les différentes catégories de conteneurs

La norme classe les différents conteneurs en deux catégories :

- les conteneurs en séquence (ou conteneurs séquentiels),
- les conteneurs associatifs.

La notion de conteneur en séquence correspond à des éléments qui sont ordonnés comme ceux d'un vecteur ou d'une liste. On peut parcourir le conteneur suivant cet ordre. Quand on insère ou qu'on supprime un élément, on le fait en un endroit qu'on a explicitement choisi.

La notion de conteneur associatif peut être illustrée par un répertoire téléphonique. Dans ce cas, on associe une valeur (numéro de téléphone, adresse...) à ce qu'on nomme une clé (ici le nom). A partir de la clé, on accède à la valeur associée. Pour insérer un nouvel élément dans ce conteneur, il ne sera théoriquement plus utile de préciser un emplacement.

Il semble donc qu'un conteneur associatif ne soit plus ordonné. En fait, pour d'évidentes questions d'efficacité, un tel conteneur devra être ordonné mais, cette fois, de façon intrinsèque, c'est-à-dire suivant un ordre qui n'est plus défini par le programme. La principale conséquence est qu'il restera toujours possible de parcourir séquentiellement les éléments d'un tel conteneur qui disposera toujours au moins d'un itérateur nommé iterator et des valeurs begin() et end(). Cet aspect peut d'ailleurs prêter à confusion, dans la mesure où certaines opérations prévues pour des conteneurs séquentiels pourront s'appliquer à des conteneurs associatifs, tandis que d'autres poseront problème. Par exemple, il n'y aura aucun risque à examiner séquentiellement chacun des éléments d'un conteneur associatif ; il y en aura manifestement, en revanche, si l'on cherche à modifier séquentiellement les valeurs d'éléments existants, puisqu'alors, on risque de perturber l'ordre intrinsèque du conteneur. Nous y reviendrons le moment venu.

3. LES GENERATEURS D'OPERATEURS

Le mécanisme de surdéfinition d'opérateurs utilisé par C++ fait que l'on peut théoriquement définir, pour une classe donnée, à la fois l'opérateur == et l'opérateur !=, de manière totalement indépendante, voir incohérente. Il en va de même pour les opérateurs <, <=, > et >=.

Mais la bibliothèque standard dispose de patrons de fonctions permettant de définir :

- l'opérateur !=, à partir de l'opérateur ==
- les opérateurs >, <= et >=, à partir de l'opérateur <.

Comme on peut s'y attendre, si a et b sont d'un type classe pour laquelle on a défini ==, != sera défini par :

a != b si !(a == b)

De la même manière, les opérateurs <=, > et >= peuvent être déduits de < par les définitions suivantes :

a > b si b < a

a <= b si ! (a > b)

a >= b si ! (a < b)

Dans ces conditions, on voit qu'il suffit de munir une classe des opérateurs == et < pour qu'elle dispose automatiquement des autres.

Bien entendu, il reste toujours possible de donner sa propre définition de l'un quelconque de ces quatre opérateurs. Elle sera alors utilisée, en tant que spécialisation d'une fonction patron.

Il est très important de noter qu'il n'existe aucun lien entre la définition automatique de <= et celle de ==. Ainsi, rien n'impose, hormis le bon sens, que a==b implique a<=b, comme le montre ce petit exemple d'école, dans lequel nous définissons l'opérateur < d'une manière artificielle et incohérente avec la définition de == :

```
#include <iostream.h>
#include <utility>          // pour les générateurs d'opérateurs
using namespace std ;       // indispensable pour utility

class point
{ int x, y ;
 public :
  point(int abs=0, int ord=0) { x=abs ; y=ord ; }
  friend int operator== (point, point) ;
  friend int operator< (point, point) ;
} ;
int operator== (point a, point b)
{ return ( (a.x == b.x) && (a.y == b.y) ) ;
}
int operator<(point a, point b)
{ return ( (a.x < b.x) && (a.y < b.y) ) ;
}
main()
```

```
{ point a(1, 2), b(3, 1) ;
  cout << "a == b : " << (a==b) << "    a != b : " << (a!=b) << "\n" ;
  cout << "a < b  : " << (a<b)  << "    a <= b : " << (a<=b) << "\n" ;
  char c ; cin >> c ;
}
```

```
a == b : 0    a != b : 1
a < b  : 0    a <= b : 1
```

Exemple de génération non satisfaisante des opérateurs !=, >, <= et >=

Remarque

Le manque de cohérence entre les définitions des opérateurs == et < est ici sans conséquence. En revanche, on verra que l'opérateur < peut intervenir, par exemple, pour ordonner un conteneur associatif ou pour trier un conteneur de type list lorsqu'on utilise la fonction membre sort. Dans ce cas, sa définition devra respecter un certain nombre de contraintes comme nous le verrons dans le paragraphe 7.

4. LES CONTENEURS DONT LES ELEMENTS SONT DES OBJETS

Le patron de classe définissant un conteneur peut être appliqué à n'importe quel type et donc, en particulier à des éléments de type classe. Dans ce cas, il ne faut pas perdre de vue que bon nombre de manipulations de ces éléments vont entraîner des appels automatiques de certaines fonctions membre.

4.1 Construction et copie d'objets

Toute construction d'un conteneur, non vide, dont les éléments sont des objets, entraîne :

- soit l'appel d'un constructeur ; il peut s'agir d'un constructeur par défaut lorsqu'aucun argument n'est nécessaire,
- soit l'appel d'un constructeur par recopie.

Par exemple, on verra que la déclaration suivante (point étant une classe) construit un vecteur de trois éléments de type point :

```
vector<point> v(3) ;  /* construction d'un vecteur de 3 points */
```

Pour chacun des trois éléments, il y aura appel d'un constructeur sans argument de point. Si l'on construit un autre vecteur, à partir de v :

```
vector<point> w (v) ;  /* ou vector v = w ; */
```

il y aura appel du constructeur par recopie de la classe vector< point> , lequel, tout naturellement, appellera le constructeur par recopie de la classe point pour chacun des trois éléments de type point à recopier.

On pourrait s'attendre à des choses comparables avec l'opérateur d'affectation dan un cas tel que :

```
w = v ;       /* le vecteur v est affecté à w */
```

Cependant, ici, les choses sont un peu moins simples. En effet, généralement, si la taille de w est suffisante, on se contentera effectivement d'appeler l'opérateur d'affectation pour tous les éléments de v (on appellera le destructeur pour les éléments excédentaires de w). En revanche, si la taille de w est insuffisante, il y aura destruction de tous ses éléments et création d'un nouveau vecteur par appel du constructeur par recopie, lequel appellera tout naturellement le constructeur par recopie de la classe point pour tous les éléments de v.

Bien entendu, pour les objets ne possédant pas de partie dynamique, les fonctions membre prévues par défaut seront satisfaisantes. Dans le cas contraire, il faudra prévoir les fonctions appropriées, ce qui sera bien sûr le cas si la classe concernée respecte le schéma de classe canonique proposé dans le chapitre IX.

Remarque

Dans les descriptions des différents conteneurs ou algorithmes, nous ne rappellerons pas ces différents points, dans la mesure où ils concernent systématiquement tous les objets.

4.2 Autres opérations

Il existe d'autres opérations que les constructions ou recopies de conteneur qui peuvent entraîner des appels automatiques de certaines fonctions membre.

L'un des exemples les plus évidents est celui de la recherche d'un élément de valeur donnée, comme le fait la fonction membre find du conteneur list. Dans ce cas, la classe concernée devra manifestement disposer de l'opérateur ==, lequel, cette fois, ne possède plus de version par défaut.

Un autre exemple réside dans les possibilités dites de "comparaisons lexicographiques" que nous examinerons dans le chapitre XIX ; nous verrons que celles-ci se fondent sur la comparaison, par l'un des opérateurs <, >, <= ou >= des différents éléments du conteneur. Manifestement, là encore, il faudra définir au moins l'opérateur < pour la classe concernée : les possibilités de génération automatique présentées ci-dessus pourront éviter les définitions des trois autres.

D'une manière générale, cette fois, compte tenu de l'aspect épisodique de ce type de besoin, nous le préciserons chaque fois que ce sera nécessaire.

5. EFFICACITE DES OPERATIONS SUR DES CONTENEURS

Pour juger de l'efficacité d'une fonction membre d'un conteneur ou d'un algorithme appliqué à un conteneur, on choisit généralement la notation dite "de Landau" (O(...)) qui se définit ainsi :

> Le temps t d'une opération est dit O(x) s'il existe une constante k telle que, dans tous les cas, on ait : t <= kx.

Comme on peut s'y attendre, le nombre N d'éléments d'un conteneur (ou d'une séquence de conteneur) pourra intervenir. C'est ainsi qu'on rencontrera typiquement :

- des opérations en O(1), c'est-à-dire pour lesquelles le temps est constant (plutôt borné par une constante, indépendante du nombre d'éléments de la séquence) ; on verra que ce sera le cas des insertions dans une liste ou des insertions en fin de vecteur ;
- des opérations en O(N), c'est-à-dire pour lesquelles le temps est proportionnel au nombre d'éléments de la séquence ; on verra que ce sera le cas des insertions en un point quelconque d'un vecteur ;
- des opérations en O(LogN)...

D'une manière générale, on ne perdra pas de vue qu'une telle information n'a qu'un caractère relativement indicatif ; pour être précis, il faudrait indiquer s'il s'agit d'un maximum ou d'une moyenne et mentionner la nature des opérations concernées. C'est d'ailleurs ce que nous ferons dans l'annexe C décrivant l'ensemble des algorithmes standard.

6. FONCTIONS, PREDICATS ET CLASSES FONCTION

6.1 Fonction unaire

Beaucoup d'algorithmes et quelques fonctions membre permettent d'appliquer une fonction donnée aux différents éléments d'une séquence (définie par un intervalle d'itérateur). Cette fonction est alors passée simplement en argument de l'algorithme, comme dans :

```
for_each(it1, it2, f) ;   /* applique la fonction f à chacun des éléments de
*/
                          /*  la séquence [it1, it2)
*/
```

Bien entendu, la fonction f doit posséder un argument du type des éléments correspondants (dans le cas contraire, on obtiendrait une erreur de compilation). Il n'est pas interdit qu'une telle fonction possède une valeur de retour mais, quoi qu'il en soit, elle ne sera pas utilisée.

Voici un exemple montrant comment utiliser cette technique pour afficher tous les éléments d'une liste :

```
main()
{ list<float> lf ;
  void affiche (float) ;
    .....
  for_each (lf.begin(), lf.end(), affiche) ; cout << "\n" ;
    .....
}
void affiche (float x) { cout << x << " " ; }
```

Bien entendu, on obtiendrait le même résultat en procédant simplement ainsi :

```
main()
{ list<float> lf ;
  void affiche (list<float>) ;
    .....
  lf.affiche() ;
    .....
}
void affiche (list<float> l)
{ list<float>::iterator il ;
  for (il=l.begin() ; il!=l.end() ; il++) cout << (*il) << " " ;
  cout << "\n" ;
}
```

6.2 Prédicats

On parle de prédicat pour caractériser une fonction qui renvoie une valeur de type bool. Compte tenu des conversions implicites qui sont mises en place automatiquement, cette valeur peut éventuellement être entière, sachant qu'alors 0 correspondra à false et que tout autre valeur correspondra à true.

On rencontrera des prédicats unaires, c'est-à-dire disposant d'un seul argument et des prédicats binaires, c'est-à-dire disposant de deux arguments de même type.

Là encore, certains algorithmes et certaines fonctions membre nécessiteront qu'on leur fournisse un prédicat en argument. Par exemple, l'algorithme find_if permet de trouver le premier élément d'une séquence vérifiant un prédicat passé en argument :

```
main()
{ list<int> l ;
  list<int>::iterator il ;
  bool impair (int) ;
    .....
  il = find_if (l.begin(), l.end(), impair) ;  /* il désigne le premier élément
de */
    .....                                      /*  l vérifiant le prédicat
impair  */
}
bool impair (int n)        /* définition du prédicat unaire impair */
{ return n%2 ; }
```

6.3 Classes et objets fonction

a) Utilisation d'objets fonction comme fonction de rappel

Nous venons de voir que certains algorithmes ou fonctions membre nécessitaient un prédicat en argument. D'une manière générale, ils peuvent nécessiter une fonction quelconque et l'on parle souvent de "fonction de rappel" pour évoquer un tel mécanisme dans lequel une fonction est amenée à appeler une autre fonction qu'on lui a transmise en argument.

La plupart du temps, cette fonction de rappel est prévue dans la définition du patron correspondant, non pas sous forme d'une fonction, mais bel et bien sous forme d'un objet de type quelconque. Les classes et les objets fonction ont été présentés dans le paragraphe 9 et nous en avions alors donné un exemple simple d'utilisation. En voici un autre qui montre l'intérêt qu'ils présentent dans le cas de patrons de fonctions. Ici, le patron de fonction essai définit une famille de fonctions recevant en argument une fonction de rappel sous forme d'un objet fonction f de type quelconque. Les exemples d'appels de la fonction essai montrent qu'on peut lui fournir, indifféremment comme fonction de rappel, soit une fonction usuelle, soit un objet fonction.

```
#include <iostream.h>
class cl_fonc             /* definition d'une classe fonction */
{ int coef ;
  public :
  cl_fonc(int n) {coef = n ;}
  int operator () (int p) {return coef*p ; }
} ;
int fct (int n)           /* definition d'une fonction usuelle */
```

```
{ return 5*n ;
}
template <class T>void essai (T f)    // definition d'essai qui reçoit en
argument
{ cout << "f(1) : " << f(1) << "\n" ; // un objet de type quelconque
  cout << "f(4) : " << f(4) << "\n" ; // et qui l'utilise comme une fonction
}
main()
{ essai (fct) ;      // appel essai en lui fournissant une fonction de rappel
usuelle
  essai (cl_fonc(3)) ; // appel essai en lui fournissant une fonction de rappel
objet
  essai (cl_fonc(7)) ; // idem
}
```

```
f(1) : 5
f(4) : 20
f(1) : 3
f(4) : 12
f(1) : 7
f(4) : 28
```

Exemple d'utilisation d'objets fonction

On voit qu'un algorithme attendant un objet fonction peut recevoir une fonction usuelle. En revanche, on notera que la réciproque est fausse. C'est pourquoi, tous les algorithmes ont prévu leurs fonctions de rappel sous forme d'objets fonction.

b) Classes fonction prédéfinies

Dans <functional>, il existe un certain nombre de patrons de classes fonction correspondant à des prédicats binaires de comparaison de deux éléments de même type. Par exemple, less<int> instancie une fonction patron correspondant à la comparaison par < (less) de deux éléments de type int. Comme on peut s'y attendre, less<point> instanciera une fonction patron correspondant à la comparaison de deux objets de type point par l'opérateur <, qui devra alors être convenablement défini dans la classe point.

Voici les différents noms de patrons existants et les opérateurs correspondants : equal_to (==), not_equal_to (!=), greater (>), less (<), greater_equal (>=), less_equal (<=).

Toutes ces classes fonction disposent d'un constructeur sans argument, ce qui leur permet d'être citées comme fonction de rappel. D'autre part, elles seront également utilisées comme argument de type dans la construction de certaines classes.

Remarque

Il existe également des classes fonction correspondant aux opérations binaires usuelles, par exemple plus<int> pour la somme de deux int. Voici les différents noms des autres patrons existants et les opérateurs correspondants : modulus (%), minus (-), times (*), divides (/). On trouve également les prédicats correspondant aux opérations logiques : logical_and (&&), logical_or (||), logical_not (!). Ces classes sont cependant d'un usage moins fréquent que celles qui ont été étudiées précédemment.

7. CONTENEURS, ALGORITHMES ET RELATION D'ORDRE

7.1 Introduction

Un certain nombre de situations nécessiteront la connaissance d'une relation permettant d'ordonner les différents éléments d'un conteneur. Citons-en quelques exemples :

- pour des questions d'efficacité, comme il a déjà été dit, les éléments d'un conteneur associatif seront ordonnés en permanence ;
- un conteneur list disposera d'une fonction membre sort, permettant de réarranger ses éléments suivant un certain ordre ;
- il existe beaucoup d'algorithmes de tri qui, eux aussi, réorganisent les éléments d'un conteneur suivant un certain ordre.

Bien entendu, tant que les éléments du conteneur concerné sont d'un type scalaire ou string, pour lequel il existe une relation naturelle (<) permettant d'ordonner les éléments, on peut se permettre d'appliquer ces différentes opérations d'ordonnancement, sans trop se poser de questions.

En revanche, si les éléments concernés sont d'un type classe qui ne dispose pas par défaut de l'opérateur <, il faudra surdéfinir convenablement cet opérateur. Dans ce cas, et comme on peut s'y attendre, cet opérateur devra respecter un certain nombre de propriétés, nécessaires au bon fonctionnement de la fonction ou de l'algorithme utilisé.

Par ailleurs, et quel que soit le type des éléments (classe, type de base...), on peut choisir d'utiliser une relation autre que celle qui correspond à l'opérateur < (par défaut ou surdéfini) :

- soit en choisissant un autre opérateur (par défaut ou surdéfini),
- soit en fournissant explicitement une fonction de comparaison de deux éléments.

Là encore, cet opérateur ou cette fonction devra respecter les propriétés évoquées que nous allons examiner maintenant.

7.2 Propriétés à respecter

Pour simplifier les notations, nous noterons toujours R, la relation binaire en question, qu'elle soit définie par un opérateur ou par une fonction. La norme précise que R doit être une relation d'ordre faible strict, laquelle se définit ainsi :

- $\forall$ a, !(a R a)
- R est transitive, c'est-à-dire que $\forall$ a, b, c, tels que : a R b et b R c, alors a R c ;
- $\forall$ a, b, c, tels que : !(a R b) et !(b R c), alors !(a R c).

On notera que l'égalité n'a pas besoin d'être définie pour que R respecte les propriétés requises.

Bien entendu, on peut sans problème utiliser les opérateurs < et > pour les types numériques ; on prendra garde, cependant, à ne pas utiliser <= ou >= qui ne répondent pas à la définition.

On peut montrer que ces contraintes définissent une relation d'ordre total, non pas sur l'ensemble des éléments concernés, mais simplement sur les classes d'équivalence induites par la relation R, une classe d'équivalence étant telle que a et b appartiennent à la même classe si l'on a à la fois !(a R b) et !(b R a). A

titre d'exemple, considérons des éléments d'un type classe (point), possédant deux coordonnées x et y ; supposons qu'on y définisse la relation R par :

p1(x1, y1) R p2(x2, y2) si x1 < x2

On peut montrer que R satisfait les contraintes requises et que les classes d'équivalence sont formées des points ayant la même abscisse.

Dans ces conditions, si l'on utilise R pour trier un conteneur de points, ceux-ci apparaîtront ordonnés suivant la première coordonnée. Cela n'est pas très grave car, dans une telle opération de tri, tous les points seront conservés. En revanche, si l'on utilise cette même relation R pour ordonner intrinsèquement un conteneur associatif de type map (dont on verra que deux éléments ne peuvent avoir de clés équivalentes), deux points de même abscisse apparaîtront comme "identiques" et un seul sera conservé dans le conteneur.

Ainsi, lorsqu'on sera amené à définir sa propre relation d'ordre, il faudra bien être en mesure d'en prévoir correctement les conséquences au niveau des opérations qui en dépendront. Notamment, dans certains cas, il faudra savoir si l'égalité de deux éléments se fonde sur l'opérateur == (surdéfini ou non), ou sur les classes d'équivalence induites par une relation d'ordre (par défaut, il s'agira alors de <, surdéfini ou non). Par exemple, l'algorithme find se fonde sur ==, tandis que la fonction membre find d'un conteneur associatif se fonde sur l'ordre intrinsèque du conteneur. Bien entendu, aucune différence n'apparaîtra avec des éléments de type numérique ou string, tant qu'on se limitera à l'ordre induit par < puisqu'alors les classes d'équivalence en question seront réduites à un seul élément.

Bien entendu, nous attirerons à nouveau votre attention sur ce point au moment voulu.

XIX. LES CONTENEURS SÉQUENTIELS

Nous avons vu, dans le précédent chapitre, que les conteneurs pouvaient se classer en deux catégories très différentes : les conteneurs séquentiels et les conteneurs associatifs ; les premiers sont ordonnés suivant un ordre imposé explicitement par le programme lui-même, tandis que les seconds le sont de manière intrinsèque. Les trois conteneurs séquentiels principaux sont les classes vector, list et deque. La classe vector généralise la notion de tableau, tandis que la classe list correspond à la notion de liste doublement chaînée. Comme on peut s'y attendre, vector disposera d'un itérateur à accès direct, tandis que list ne disposera que d'un itérateur bidirectionnel. Quant à la classe deque, on verra qu'il s'agit d'une classe intermédiaire entre les deux précédentes dont la présence ne se justifie que pour des questions d'efficacité.

Nous commencerons par étudier les fonctionnalités communes à ces trois conteneurs : construction, affectation globale, initialisation par un autre conteneur, insertion et suppression d'éléments, comparaisons... Puis nous examinerons en détail les fonctionnalités spécifiques à chacun des conteneurs vector, deque et list. Nous terminerons par une brève description des trois adaptateurs de conteneurs que sont stack, queue et priority_queue.

1. FONCTIONNALITES COMMUNES AUX CONTENEURS VECTOR, LIST ET DEQUE

Comme tous les conteneurs, vector, list et deque sont de taille dynamique, c'est-à-dire susceptibles de varier au fil de l'exécution. Malgré leur différence de nature, ces trois conteneurs possèdent des fonctionnalités communes que nous allons étudier ici. Elles concernent :

- leur construction,
- l'affectation globale,
- leur comparaison,
- l'insertion de nouveaux éléments ou la suppression d'éléments existants.

1.1 Construction

Les trois classes vector, list et deque disposent de différents constructeurs : conteneur vide, avec nombre d'éléments donné, à partir d'un autre conteneur.

Construction d'un conteneur vide

L'appel d'un constructeur sans argument construit un conteneur vide, c'est-à-dire ne comportant aucun élément :

```
list<float> lf ;      /* la liste lf est construite vide ; lf.size() vaudra 0 ;
*/
                      /*  et lf.begin() == lf.end()
*/
```

Construction avec un nombre donné d'éléments

De façon comparable à ce qui se passe avec la déclaration d'un tableau classique, l'appel d'un constructeur avec un seul argument entier n construit un conteneur comprenant n éléments. En ce qui concerne l'initialisation de ces éléments, elle est régie par les règles habituelles dans le cas d'éléments de type standard (0 pour la classe statique, indéterminé sinon). Lorsqu'il s'agit d'éléments de type classe, ils sont tout naturellement initialisés par appel d'un constructeur sans argument.

```
list<float> lf(5) ;   /* lf est construite avec 5 éléments de type float     */
                      /*    lf.size() vaut 5                                 */
vector<point> vp(5) ; /* vp est construit avec 5 éléments de type point      */
                      /* initialisés par appel du constructeur sans argument */
```

Construction avec un nombre donné d'éléments initialisés à une valeur imposée

Le premier argument du constructeur fournit le nombre d'éléments, le second argument en fournit la valeur :

```
list<int> li(5, 999) ;   /* li est construite avec 5 éléments de type int
*/
                          /* ayant tous la valeur 999
*/
point a(3, 8) ;           /* on suppose que point est une classe...
*/
list<point> lp (10, a) ;  /* lp est construite avec 10 points ayant tous
*/
                          /* la valeur de a : il y a appel du constructeur par
*/
                          /* recopie (éventuellement par défaut) de point
*/
```

Construction à partir d'une séquence

On peut construire un conteneur à partir d'une séquence d'éléments de même type. Dans ce cas, on fournit simplement au constructeur deux arguments représentant les bornes de l'intervalle correspondant. Voici des exemples utilisant des séquences de conteneur de type list< point> :

```
list<point> lp(6) ;          /* liste de 6 points */
   .....
vector<point> vp (lp.begin(), lp.end()) ;    /* construit un vecteur de points
*/
                  /* en recopiant les points de la liste lp  ; le constructeur
*/
                    /* par recopie de point sera appelé pour chacun des points
*/
list<point> lpi (lp.rbegin(), lp.rend()) ;   /* construit une liste obtenue en
*/
                                 /* inversant l'ordre des points de la liste lp
*/
```

Ici, les séquences correspondaient à l'ensemble du conteneur ; il s'agit de la situation la plus usuelle, mais rien n'empêcherait d'utiliser des intervalles d'itérateurs quelconques, pour peu que la seconde borne soit accessible à partir de la première.

Voici un autre exemple de construction de conteneurs, à partir de séquences de valeurs issues d'un tableau classique, utilisant des intervalles définis par des pointeurs :

```
int t[6] = { 2, 9, 1, 8, 2, 11 } ;
vector<int> vi(t, t+6) ;  /* construit un vecteur formé des 6 valeurs du
tableau t */
vector<int> vi2(t+1, t+5) ; /* construit un vecteur formé des valeurs t[1] à
t[5]  */
```

Dans le premier cas, si l'on souhaite une formulation indépendante de la taille effective de t, on pourra procéder ainsi :

```
int t[] = { ..... } ;                          /* nombre quelconque de valeurs
*/
vector<int> vi(t, t + sizeof(t)/sizeof(int)) ; /* qui seront recopiées dans vi
*/
```

Construction à partir d'un autre conteneur de même type

Il s'agit d'un classique constructeur par recopie qui, comme on peut s'y attendre, appelle le constructeur de recopie des éléments concernés lorsqu'il s'agit d'objets.

```
vector<int> vi1 ;   /* vecteur d'entiers */
   .....
vector<int> vi2(vi1) ;   /* ou encore vector<int> vi2 = vi1 ;  */
```

1.2 Modifications globales

Les trois classes vector, deque et list définissent convenablement l'opérateur d'affectation ; de plus, elles proposent une fonction membre assign, comportant plusieurs définitions, ainsi qu'une fonction clear.

a) Opérateur d'affectation

On peut affecter un conteneur d'un type donné à un autre conteneur de même type, c'est-à-dire ayant le même nom de patron et le même type d'éléments. Bien entendu, il n'est nullement nécessaire que le nombre d'éléments de chacun des conteneurs soit le même. Voici quelques exemples :

```
vector<int> vi1 (...), vi2 (...) ;
vector<float> vf (...) ;
   .....
vi1 = vi2 ; /* correct, quel que soit le nombre d'éléments de vi1 et de vi2
*/
            /* le contenu de vi1 est remplacé par celui de vi2 qui reste
inchangé */
vf = vi1 ;  /* incorrect (refusé en compilation) : les éléments de vf et de vi1
*/
            /* ne sont pas du même type
*/
```

Voici un autre exemple avec un conteneur dont les éléments sont des objets :

```
vector<point> vp1 (....), vp2 (...) ;
   .....
vp1 = vp2 ;
```

Dans ce cas, comme nous l'avons déjà fait remarquer dans le paragraphe 4.1 du précédent chapitre, il existe deux façons de parvenir au résultat escompté, suivant les tailles relatives des vecteurs vp1 et vp2, à savoir, soit l'utilisation du constructeur par recopie de la classe point, soit l'utilisation de l'opérateur d'affectation de la classe point.

b) La fonction membre assign

Alors que l'affectation n'est possible qu'entre conteneurs de même type, la fonction assign permet d'affecter, à un conteneur existant, les éléments d'une séquence définie par un intervalle [debut, fin), à condition que les éléments désignés soient du type voulu (et pas seulement d'un type compatible par affectation) :

assign (début, fin) /* fin doit être accessible depuis début */

Il existe également une autre version permettant d'affecter à un conteneur, un nombre donné d'éléments ayant une valeur imposée :

assign (nb_fois, valeur)

Dans les deux cas, les éléments existants seront remplacés par les éléments voulus, comme s'il y avait eu affectation.

```
point a (...) ;
list<point> lp (...) ;
vector<point> vp (...) ;
   .....
lp.assign (vp.begin(), vp.end()) ;    /* on a maintenant : lp.size() =
vp.size() */
vp.assign (10, a) ;                   /* on a maintenant : vp.size()=10
*/


char t[] = {"hello"} ;
list<char> lc(7, 'x') ;   /* la liste contient :            x, x, x, x, x, x,
x  */
   .....
lc.assign(t, t+4) ;       /* la liste contient maintenant :  h, e, l, l, o
*/
lc.assign(3, 'z') ;       /* la liste contient maintenant :  z, z, z
*/
```

c) La fonction clear

La fonction clear() vide le conteneur de son contenu.

```
vector<point> vp(10) ;  /* vp.size() = 0 */
   .....
vp.clear () ;      /* appel du destructeur de chacun des points de vp */
                   /* maintenant vp.size() = 0                        */
```

d) La fonction swap

La fonction membre swap (conteneur) permet d'échanger le contenu de deux conteneurs de même type. Par exemple :

```
vector<int> v1, v2 ;
   .....
v1.swap(v2) ;   /* ou :  v2.swap(v1) ;  */
```

L'affectation précédente sera plus efficace que la démarche traditionnelle :

```
vector<int> v3=v1 ;
v1=v2 ;
v2=v3 ;
```

Remarque

Comme on peut le constater, les possibilités de modifications globales d'un conteneur sont similaires à celles qui sont offertes au moment de la construction, la seule possibilité absente étant l'affectation d'un nombre d'éléments donnés, éventuellement non initialisés.

1.3 Comparaison de conteneurs

Les trois conteneurs vector, deque et list disposent des opérateurs == et < ; par le biais des générations automatiques d'opérateurs, ils disposent donc également de !=, <=, > et >=. Le rôle de == correspond à ce qu'on attend d'un tel opérateur, tandis que celui de < s'appuie sur ce que l'on nomme parfois une comparaison lexicographique, analogue à celle qui permet de classer des mots par ordre alphabétique.

a) L'opérateur ==

Il ne présente pas de difficultés particulières. Si c1 et c2 sont deux conteneurs de même type, leur comparaison par == sera vraie s'ils ont la même taille et si les éléments de même rang sont égaux.

On notera cependant que si les éléments concernés sont de type classe, il sera nécessaire que cette dernière dispose elle-même de l'opérateur ==.

b) L'opérateur <

Il effectue une comparaison lexicographique des éléments des deux conteneurs. Pour ce faire, il compare les éléments de même rang, par l'opérateur <, en commençant par les premiers, s'ils existent. Il s'interrompt dès que l'une des conditions suivantes est réalisée :

- fin de l'un des conteneurs atteinte ; le résultat de la comparaison est vrai,
- comparaison de deux éléments fausse ; le résultat de la comparaison des conteneurs est alors faux.

Si un seul des deux conteneurs est vide, il apparaît comme < à l'autre. Si les deux conteneurs sont vides, aucun n'est inférieur à l'autre (ils sont égaux).

On notera, là encore, que si les éléments concernés sont de type classe, il sera nécessaire que cette dernière dispose elle-même d'un opérateur < approprié.

c) Exemples

Avec ces déclarations :

```
int t1[] = {2, 5, 2, 4, 8 } ;
int t2[] = {2, 5, 2, 8 } ;
vector<int> v1 (t1, t1+5) ;    /* v1 contient : 2 5 2 4 8 */
vector<int> v2 (t2, t2+4) ;    /* v2 contient : 2 5 2 8   */
vector<int> v3 (t2, t2+3) ;    /* v3 contient : 2 5 2     */
vector<int> v4 (v3) ;          /* v4 contient : 2 5 2     */
vector<int> v5 ;               /* v5 est vide             */
```

Voici quelques comparaisons possibles et la valeur correspondante :

```
v2 < v1  /* faux */        v3 < v2   /* vrai */        v3 < v4  /* faux */
v4 < v3  /* faux */        v3 == v4  /* vrai */        v4 > v5  /* vrai */
v5 > v5  /* faux */        v5 < v5   /* faux */
```

1.4 Insertion ou suppression d'éléments

Chacun des trois conteneurs vector, deque et list dispose naturellement de possibilités d'accès à un élément existant, soit pour en connaître la valeur, soit pour la modifier. Comme ces possibilités varient quelque peu d'un conteneur à l'autre, elles seront décrites dans les paragraphes ultérieurs. Par ailleurs, ces trois conteneurs (comme tous les conteneurs) permettent des modifications dynamiques fondées sur des insertions de nouveaux éléments ou des suppressions d'éléments existants. On notera que de telles possibilités n'existaient pas dans le cas d'un tableau classique, alors qu'elles existent pour le conteneur vector, même si, manifestement, elles sont davantage utilisées dans le cas d'une liste.

Rappelons toutefois que, bien qu'en théorie, les trois conteneurs offrent les mêmes possibilités d'insertions et de suppressions, leur efficacité sera différente d'un conteneur à un autre. Nous verrons dans les paragraphes suivants que, dans une liste, elles seront toujours en O(1), tandis que dans les conteneurs vector et deque, elles seront en O(N), excepté lorsqu'elles auront lieu en fin de vector ou en début ou en fin de deque où elles se feront en O(1) ; dans ces derniers cas, on verra d'ailleurs qu'il existe des fonctions membre spécialisées.

a) Insertion

La fonction insert permet d'insérer :

- une valeur avant une position donnée :

```
insert (position, valeur)    /* insère valeur avant l'élément pointé par position */
                             /* fournit un itérateur sur l'élément inséré         */
```

- n fois une valeur donnée, avant une position donnée :

```
insert (position, nb_fois, valeur)   /* insère nb_fois valeur, avant l'élément  */
                                     /*  pointé par position                    */
                                     /* fournit un itérateur sur l'élément inséré */
```

- les éléments d'un intervalle, à partir d'une position donnée :

```
insert (debut, fin, position)   /* insère les valeurs de l'intervalle [debut,  fin) , */
                                /* avant l'élément pointé par position              */
```

En voici quelques exemples :

```
list<double> ld ;
list<double>::iterator il ;
   .....    /* on suppose que il pointe correctement dans la liste ld */
```

```
ld.insert (il, 2.5) ;          /* insère 2.5 dans ld, avant l'élément pointé par
il */
ld.insert (ld.begin(), 6.7) ; /* insère 6.7 au début de ld
*/
ld.insert (ld.end(),   3.2) ; /* insère 3.2 en fin de ld
*/
   .....
ld.insert(il, 10, -1) ;        /* insère 10 fois -1, avant l'élément pointé par
il  */
   .....
vector<double> vd (...) ;
ld.insert(ld.begin(), vd.begin(), vd.end()) ;   /* insère tous les éléments de
vd  */
                                                /* en début de la liste ld
*/
```

b) Suppression

La fonction erase permet de supprimer :

- un élément de position donnée :

  erase (position) /* supprime l'élément désigné par position - fournit un itérateur */
  /* sur l'élément suivant ou sur la fin de la séquence */

- les éléments d'un intervalle :

  erase (début, fin) /*supprime les valeurs de l'intervalle [début, fin) - fournit un */
  /* itérateur sur l'élément suivant ou sur la fin de la séquence */

En voici quelques exemples :

```
list<double> ld ;
list<double>::iterator il1, il2 ;
   .....    /* on suppose que il1 et il2 pointent correctement dans */
            /* la liste ld et que il2 est accessible à partir de il1*/
ld.erase(il1, il2) ;    /* supprime les éléments de l'intervalle [il1, il2) */
ld.erase(ld.begin()) ;                /* supprime l'élément de début de ld */
```

Remarques

1) Les deux fonctions erase renvoient la valeur de l'itérateur suivant le dernier élément supprimé s'il en existe un ou sinon, la valeur end(). Voyez par exemple, la construction suivante, dans laquelle il est un itérateur, de valeur convenable, sur une liste d'entiers ld :

   ```
   while (il = ld.erase(il) != ld.end()) ;
   ```

   Elle est équivalente à :

   ```
   erase (il, ld.end()) ;
   ```

2) Les conteneurs séquentiels ne sont pas adaptés à la recherche de valeurs données ou à leur suppression. Il n'existera d'ailleurs aucune fonction membre à cet effet, contrairement à ce qui se produira avec les conteneurs associatifs. Il n'en reste pas moins qu'une telle recherche peut toujours se faire à l'aide d'un algorithme standard tel que find ou find_if, mais au prix d'une efficacité médiocre (en O(N)).

c) Cas des insertions/suppressions en fin : pop_back et push_back

Si l'on s'en tient aux possibilités générales présentées ci-dessus, on constate que s'il est possible de supprimer le premier élément d'un conteneur en appliquant erase à la position begin(), il n'est pas possible de supprimer le dernier élément d'un conteneur, en appliquant erase à la position end(). Un tel résultat peut toutefois s'obtenir en appliquant erase à la position rbegin(). Quoi qu'il en soit, comme l'efficacité de cette suppression est en O(1) pour les trois conteneurs, il existe une fonction membre spécialisée pop_back() qui réalise cette opération ; si c est un conteneur, c.pop_back() est équivalente à c.erase(c.rbegin()).

D'une manière semblable, et bien que ce ne soit guère indispensable, il existe une fonction spécialisée d'insertion en fin push_back. Si c est un conteneur, c.push_back(valeur) est équivalent à c.insert (c.end(), valeur).

2. LE CONTENEUR VECTOR

Il reprend la notion usuelle de tableau en autorisant un accès direct à un élément quelconque avec une efficacité en O(1), c'est-à-dire indépendante du nombre de ses éléments. Cet accès peut se faire soit par le biais d'un itérateur à accès direct, soit de façon plus classique, par l'opérateur [ ] ou par la fonction membre at. Mais il offre un cadre plus général que le tableau puisque :

- la taille, c'est-à-dire le nombre d'éléments, peut varier au fil de l'exécution (comme celle de tous les conteneurs) ;
- on peut effectuer toutes les opérations de construction, d'affectation et de comparaisons décrites aux paragraphes 1.1, 1.2 et 1.3 ;
- on dispose des possibilités générales d'insertion ou de suppressions décrites au paragraphe 1.4 (avec, cependant, une efficacité en O(N) dans le cas général).

Ici, nous nous contenterons d'examiner les fonctionnalités spécifiques de la classe vector, qui viennent donc en complément de celles qui sont examinées dans le paragraphe 1.

2.1 Accès aux élément existants

On accède aux différents éléments d'un vecteur, aussi bien pour en connaître la valeur que pour la modifier, de différentes manières : par itérateur (iterator ou reverse_iterator) ou par indice (opérateur [ ] ou fonction membre at). En outre, l'accès au dernier élément peut se faire par une fonction membre appropriée back. Dans tous les cas, l'efficacité de cet accès est en O(1), ce qui constitue manifestement le point fort de ce type de conteneur.

Accès par itérateur

Les itérateurs iterator et reverse_iterator d'un conteneur de type vector sont à accès direct. Si, par exemple, iv est une variable de type vector< int> ::iterator, une expression telle que iv+i a alors un sens : elle désigne l'élément du vecteur v, situé i éléments plus loin que celui qui est désigné par iv, à condition que la valeur de i soit compatible avec le nombre d'éléments de v.

L'itérateur iv peut, bien sûr, comme tout itérateur bidirectionnel, être incrémenté ou décrémenté par ++ ou --. Mais, comme il est à accès direct, il peut également être incrémenté ou décrémenté d'une quantité quelconque, comme dans :

```
iv += n ;  iv -= p ;
```

Voici un petit exemple d'école

```
vector<int> vi(10) ;                    /* vecteur de 10 éléments
*/
vector<int>::iterator iv = v.begin() ;  /* iv pointe sur le premier élément de
v  */
  .....
iv = vi.begin() ; *iv=0 ;     /* place la valeur 0 dans le premier élément de vi
*/
iv+=3 ; *iv=30 ;              /* place la valeur 30 dans le quatrième élément de
vi */
iv = vi.end()-2 ;  *iv=70 ;  /* place la valeur 90 dans le huitième élément de
vi  */
```

Accès par indice

L'opérateur [ ] est, en fait, utilisable de façon naturelle. Si v est de type vector, l'expression v[i] est une référence à l'élément de rang i, de sorte que les deux instructions suivantes sont équivalentes :

```
v[i] = ... ;       *(v.begin()+i) = ... ;
```

Mais il existe également une fonction membre at telle que v.at(i) soit équivalente à v[i]. Sa seule raison d'être est de générer une exception out_of_range en cas d'indice incorrect, ce que l'opérateur [ ] ne fait théoriquement pas. Bien entendu, en contrepartie, at est légèrement moins rapide que l'opérateur [ ].

L'exemple d'école précédent peut manifestement s'écrire plus simplement :

```
vi[0] = 0 ;                                    /* ou :  vi.at(0) = 0 ;   */
vi[3] = 30 ;                                   /* ou :  vi.at(3) = 30 ;  */
vi[7] = 70 ;   /* ou : vi[vi.size()-2] = 70 ;      ou :  vi.at(7) = 70 ;  */
```

Il est généralement préférable d'utiliser les indices plutôt que les itérateurs dont le principal avantage réside dans l'homogénéisation de notation avec les autres conteneurs.

Cas de l'accès au dernier élément

Comme le vecteur est particulièrement adapté aux insertions ou aux suppressions en fin, il existe une fonction membre back qui permet d'accéder directement au dernier élément.

```
vector<int> v(10) ;
   .....
v.back() = 25 ;   /* équivalent, quand v est de taille 10, à :    v[9] = 25 ;
*/
                  /* équivalent, dans tous les cas, à :   v[v.size()-1] = 25
*/
```

On notera bien que cette fonction se contente de fournir une référence à un élément existant. Elle ne permet en aucun cas des insertions ou des suppressions en fin, lesquelles sont étudiées ci-dessous.

2.2 Insertions et suppressions

Le conteneur vector dispose des possibilités générales d'insertion et de suppression décrites au paragraphe 1.4. Toutefois, leur efficacité est médiocre, puisqu'en O(N), alors que, dans le cas des listes, elle sera en

O(1). C'est là le prix à payer pour disposer d'accès aux éléments existant en O(1). En revanche, nous avons vu que, comme les deux autres conteneurs, vector disposait de fonctions membre d'insertion ou de suppression du dernier élément, dont l'efficacité est en O(1) :

- la fonction push_back(valeur) pour insérer un nouvel élément en fin,
- la fonction pop_back() pour supprimer le dernier élément.

Voici un petit exemple d'école :

```
vector<int> v(5, 99) ;   _     /* vecteur de 5 éléments de valeur 99  v.size()
= 5 */
v.push_back(10) ;    /* ajoute un élément de valeur 10 : v.size() = 6 et v[5] =
10 */
                     /* ici, v[6] n'existe pas
*/
v.push_back(20) ;    /* ajoute un élément de valeur 20 : v.size() = 7 et v[6] =
20 */
v.pop_back() ;       /* supprime le dernier élément :    v.size() = 6
*/
```

2.3 Gestion de l'emplacement mémoire

a) Introduction

La norme n'impose pas explicitement la manière dont une implémentation doit gérer l'emplacement alloué à un vecteur. Cependant, comme nous l'avons vu, elle impose des contraintes d'efficacité à certaines opérations, ce qui, comme on s'en doute, limite sévèrement la marge de manœuvre de l'implémentation.

Par ailleurs, la classe vector dispose d'outils fournissant des informations relatives à la gestion des emplacements mémoire et permettant, éventuellement, d'intervenir dans leur allocation. Bien entendu, le rôle de tels outils est plus facile à appréhender lorsque l'on connaît la manière exacte dont une implémentation gère un vecteur.

Enfin, la norme prévoit que, suite à certaines opérations, des références ou des valeurs d'itérateurs peuvent devenir invalides, c'est-à-dire inutilisables pour accéder aux éléments correspondants. Là encore, il est plus facile de comprendre les règles imposées si l'on connaît la manière dont l'implémentation gère les emplacements mémoire.

Or, précisément, les implémentations actuelles allouent toujours l'emplacement d'un vecteur en un seul bloc. Même si ce n'est pas la seule solution envisageable, c'est certainement la plus plausible.

b) Invalidation d'itérateurs ou de références

Un certain nombre d'opérations sur un vecteur entraînent l'invalidation des itérateurs ou des références sur certains des éléments de ce vecteur. Les éléments concernés sont exactement ceux auxquels on peut s'attendre dans le cas où l'emplacement mémoire est géré en un seul bloc, à savoir :

- tous les éléments, en cas d'augmentation de la taille ; en effet, il se peut qu'une recopie de l'ensemble du vecteur ait été nécessaire ; on verra toutefois qu'il est possible d'éviter certaines recopies en réservant plus d'emplacements que nécessaire ;
- tous les éléments, en cas d'insertion d'un élément ; la raison en est la même ;
- les éléments situés à la suite d'un élément supprimé, ainsi que l'élément supprimé (ce qui va de soi !) ; ici, on voit que seuls les éléments situés à la suite de l'élément supprimé ont dû être déplacés.

c) Outils de gestion de l'emplacement mémoire d'un vecteur

La norme propose un certain nombre d'outils fournissant des informations concernant l'emplacement mémoire alloué à un vecteur et permettant, éventuellement, d'intervenir dans son allocation. Comme on l'a dit en introduction, le rôle de ces outils est plus facile à appréhender si l'on fait l'hypothèse que l'emplacement d'un vecteur est toujours alloué sous forme d'un bloc unique.

On a déjà vu que la fonction size() permettait de connaître le nombre d'éléments d'un vecteur. Mais il existe une fonction voisine, capacity(), qui fournit la taille potentielle du vecteur, c'est-à-dire le nombre d'éléments qu'il pourra accepter, sans avoir à effectuer de nouvelle allocation. Dans le cas usuel où le vecteur est alloué sous forme d'un seul bloc, cette fonction en fournit simplement la taille (l'unité utilisée restant l'élément du vecteur). Bien entendu, à tout instant, on a toujours capacity() >= size(). La différence capacity()-size() permet de connaître le nombre d'éléments qu'on pourra insérer dans un vecteur sans qu'une réallocation de mémoire soit nécessaire.

Mais une telle information ne serait guère intéressante si l'on ne pouvait pas agir sur cette allocation. Or, la fonction membre reserve(taille) permet précisément d'imposer la taille minimale de l'emplacement alloué à un vecteur à un moment donné. Bien entendu, l'appel de cette fonction peut très bien amener à une recopie de tous les éléments du vecteur en un autre emplacement. Cependant, une fois ce travail accompli, tant que la taille du vecteur ne dépassera pas la limite allouée, on est assuré de limiter au maximum les recopies d'éléments en cas d'insertion ou de suppression. En particulier, en cas d'insertion d'un nouvel élément, les éléments situés avant ne seront pas déplacés et les références ou itérateurs correspondants resteront valides.

Par ailleurs, la fonction max_size() permet de connaître la taille maximale qu'on peut allouer au vecteur, à un instant donné.

Enfin, il existe une fonction resize(taille), peu usitée, qui permet de modifier la taille effective du vecteur, aussi bien dans le sens de l'accroissement que dans celui de la réduction. Attention, il ne s'agit plus, ici, comme avec reserve, d'agir sur la taille de l'emplacement alloué, mais, bel et bien, sur le nombre d'éléments du vecteur. Si l'appel de resize conduit à augmenter la taille du vecteur, on lui insère, en fin, de nouveaux éléments. Si, en revanche, l'appel conduit à diminuer la taille du vecteur, on supprime, en fin, le nombre d'éléments voulus avec, naturellement, appel de leur destructeur, s'il s'agit d'objets.

2.4 Exemple

Voici un exemple complet de programme illustrant les principales fonctionnalités de la classe vector que nous venons d'examiner dans ce paragraphe et dans le précédent. Nous y avons adjoint une recherche de valeur par l'algorithme find qui ne sera présenté qu'ultérieurement, mais dont la signification est assez évidente : rechercher une valeur donnée.

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ void affiche (vector<int>) ;
  int i ;
  int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } ;
  vector<int> v1(4, 99) ;  // vecteur de 4 entiers egaux à 99
  vector<int> v2(7, 0) ;   // vecteur de 7 entiers
  vector<int> v3(t, t+6) ; // vecteur construit a partir de t
```

```
    cout << "V1 init = " ; affiche(v1)  ;
    for (i=0 ; i<v2.size() ; i++) v2[i] = i*i ;
    v3 = v2 ;
    cout << "V2 = " ; affiche(v2) ;
    cout << "V3 = " ; affiche(v3) ;
    v1.assign (t+1, t+6) ; cout << "v1 apres assign : " ; affiche(v1) ;
    cout << "dernier element de v1 : " << v1.back() << "\n" ;
    v1.push_back(99) ; cout << "v1 apres push_back : " ; affiche(v1) ;
    v2.pop_back() ; cout << "v2 apres pop_back : " ; affiche(v2) ;
    cout << "v1.size() : " << v1.size() << "   v1.capacity() : "
         << v1.capacity() << "   V1.max_size() : " << v1.max_size() << "\n" ;
    vector<int>::iterator iv ;
    iv = find (v1.begin(), v1.end(), 16) ; // recherche de 16 dans v1
    if (iv != v1.end()) v1.insert (iv, v2.begin(), v2.end()) ;
                // attention, ici iv n'est plus utilisable
    cout << "v1 apres insert : " ; affiche(v1) ;
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++) cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
V1 init = 99 99 99 99
V2 = 0 1 4 9 16 25 36
V3 = 0 1 4 9 16 25 36
v1 apres assign : 2 3 4 5 6
dernier element de v1 : 6
v1 apres push_back : 2 3 4 5 6 99
v2 apres pop_back : 0 1 4 9 16 25
v1.size() : 6   v1.capacity() : 10   V1.max_size() : 1073741823
v1 apres insert : 2 3 4 5 6 99
```

Exemple d'utilisation de la classe vector

2.5 Cas particulier des vecteurs de booléns

La norme prévoit l'existence d'une spécialisation du patron vector, lorsque son argument est de type bool. L'objectif principal est de permettre à l'implémentation d'optimiser le stockage sur un seul bit des informations correspondant à chaque élément. Les fonctionnalités de la classe vector<bool> sont donc celles que nous avons étudiées précédemment. Il faut cependant lui adjoindre une fonction membre flip destinée à inverser tous les bits d'un tel vecteur.

3. LE CONTENEUR DEQUE

3.1 Présentation générale

Le conteneur deque offre des fonctionnalités assez voisines de celles d'un vecteur. En particulier, il permet toujours l'accès direct en O(1) à un élément quelconque, tandis que les suppressions et insertions en un point quelconque restent en O(N). En revanche, il offre, en plus de l'insertion ou suppression en fin, une insertion

ou suppression en début, également en O(1), ce que ne permettait pas le vecteur. En fait, il ne faut pas en conclure pour autant que deque est plus efficace que vector car cette possibilité supplémentaire se paye à différents niveaux :

- une opération en O(1) sur un conteneur de type deque sera moins rapide que la même opération, toujours en O(1) sur un conteneur de type vector ;
- certains outils de gestion de l'emplacement mémoire d'un conteneur de type vector, n'existent plus pour un conteneur de type deque ; plus précisément, on disposera bien de size() et de max_size(), mais plus de capacity et de reserve.

Là encore et comme nous l'avons fait remarquer au paragraphe 2.3, la norme n'impose pas explicitement la manière de gérer l'emplacement mémoire d'un conteneur de type deque ; néanmoins, les choses deviennent beaucoup plus compréhensibles si l'on admet que, pour satisfaire aux contraintes imposées, il n'est pas raisonnable d'allouer un deque en un seul bloc mais plutôt sous forme de plusieurs blocs contenant chacun un ou, généralement, plusieurs éléments. Dans ces conditions, on voit bien que l'insertion ou la suppression en début de conteneur ne nécessitera plus le déplacement de l'ensemble des autres éléments, comme c'était le cas avec un vecteur, mais seulement de quelques-uns d'entre eux. En revanche, plus la taille des blocs sera petite, plus la rapidité de l'accès direct (bien que toujours en O(1)) diminuera. Au contraire, les insertions et les suppressions, bien qu'ayant une efficacité en O(N), seront d'autant plus performantes que les blocs seront petits.

Si l'on fait abstraction de ces différences de performances, les fonctionnalités de deque sont celles de vector, auxquelles il faut, tout naturellement, ajouter les fonctions spécialisées concernant le premier élément :

- front(), pour accéder au premier élément ; elle complète la fonction back permettant l'accès au dernier élément ;
- push_front(valeur), pour insérer un nouvel élément en début ; elle complète la fonction push_back() ;
- pop_front(), pour supprimer le premier élément ; elle complète la fonction pop_back().

Les règles d'invalidation des itérateurs et des références restent exactement les mêmes que celles de la classe vector, même si, dans certains cas, elles peuvent apparaître très contraignantes. Par exemple, si un conteneur de type deque est implémenté sous forme de 5 blocs différents, il est certain que l'insertion en début, n'invalidera que les itérateurs sur des éléments du premier bloc qui sera le seul soumis à une recopie ; mais, en pratique, on ne pourra jamais profiter de cette remarque ; d'ailleurs, on ne connaîtra même pas la taille des blocs !

D'une manière générale, le conteneur deque est beaucoup moins utilisé que les conteneurs vector et list qui possèdent des fonctionnalités bien distinctes. Il peut s'avérer intéressant dans des situations de pile de type FIFO (First In, First Out) où il est nécessaire d'introduire des informations à une extrémité, et de les recueillir à l'autre. En fait, dans ce cas, si l'on n'a plus besoin d'accéder directement aux différents éléments, il est préférable d'utiliser l'adaptateur de conteneur queue dont nous parlerons au paragraphe 5.

3.2 Exemple

Voici un petit exemple d'école illustrant quelques-unes des fonctionnalités du conteneur deque :

```
#include <iostream.h>
#include <deque>
using namespace std ;
main()
{
```

```
    void affiche (deque<char>) ;
    char mot[] = {"xyz"} ;
    deque<char> pile(mot, mot+3) ; affiche(pile) ;
    pile.push_front('a') ;         affiche(pile) ;
    pile[2] = '+' ;
    pile.push_front('b') ;
    pile.pop_back() ;              affiche(pile) ;
    deque<char>::iterator ip ;
    ip = find (pile.begin(), pile.end(), 'x') ;
    pile.erase(pile.begin(), ip) ; affiche(pile) ;
}
void affiche (deque<char> p)
{   int i ;
    for (i=0 ; i<p.size() ; i++) cout << p[i] << " " ;
    cout << "\n" ;
}
```

```
x y z
a x y z
b a x +
x +
```

Exemple d'utilisation de la classe deque

4. LE CONTENEUR LIST

Le conteneur list correspond au concept de liste doublement chaînée, ce qui signifie qu'on y disposera d'un itérateur bidirectionnel permettant de parcourir la liste à l'endroit ou à l'envers. Cette fois, les insertions ou suppressions vont se faire avec une efficacité en O(1), quelle que soit leur position, ce qui constitue l'atout majeur de ce conteneur par rapport aux deux classes précédentes vector et deque. En contrepartie, le conteneur list ne dispose plus d'un itérateur à accès direct. Rappelons que toutes les possibilités exposées dans le paragraphe 1 s'appliquent aux listes ; nous ne les reprendrons donc pas ici.

4.1 Accès aux éléments existants

Les conteneurs vector et deque permettaient d'accéder aux éléments existants de deux manières : par itérateur ou par indice ; en fait, il existait un lien entre ces deux possibilités parce que les itérateurs de ces classes étaient à accès direct. Le conteneur list offre toujours les itérateurs iterator et reverse_iterator mais, cette fois, ils sont seulement bidirectionnels. Si it désigne un tel itérateur, on pourra toujours consulter l'élément pointé par la valeur de l'expression *it, ou le modifier par une affectation de la forme :

```
*it = ... ;
```

L'itérateur it pourra être incrémenté par ++ ou --, mais il ne sera plus possible de l'incrémenter d'une quantité quelconque. Ainsi, pour accéder une première fois à un élément d'une liste, il aura fallu obligatoirement la parcourir depuis son début ou depuis sa fin, élément par élément, jusqu'à l'élément concerné et ceci, quel que soit l'intérêt qu'on peut attacher aux éléments intermédiaires.

La classe list dispose des fonctions front() et back(), avec la même signification que pour la classe deque : la première est une référence au premier élément, la seconde est une référence au dernier élément :

```
list<int> l () ;
   .....
if (l.front()=99) l.front=0 ;    /* si le premier élément vaut 99, */
                                 /* on lui donne la valeur 0       */
```

On ne confondra pas la modification de l'un de ces éléments, opération qui ne modifie pas le nombre d'éléments de la liste, avec l'insertion en début ou en fin de liste qui modifie le nombre d'éléments de la liste.

4.2 Insertions et suppressions

Le conteneur list dispose des possibilités générales d'insertion et de suppression procurées par les fonctions insert et erase et décrites au paragraphe 1.4. Mais, cette fois, leur efficacité est toujours en O(1), ce qui n'était pas le cas, en général, des conteneurs vector et deque. On dispose également des fonctions spécialisées d'insertion en début push_front(valeur) ou en fin push_back(valeur) ou de suppression en début pop_front() ou en fin pop_back(), rencontrées dans les classes vector et deque.

En outre, la classe list dispose de fonctions de suppressions conditionnelles que ne possédaient pas les conteneurs précédents :

- suppression de tous les éléments ayant une valeur donnée,
- suppression des éléments satisfaisant à une condition donnée.

a) Suppression des éléments de valeur donnée

remove(valeur) /* supprime tous éléments égaux à valeur */

Comme on peut s'y attendre, cette fonction se fonde sur l'opérateur == qui doit donc être défini dans le cas où les éléments concernés sont des objets :

```
int t[] = {1, 3, 1, 6, 4, 1, 5, 2, 1 }
list<int> li(t, t+9) ;   /* li contient :            1, 3, 1, 6, 4, 1, 5, 2, 1
*/
li.remove (1) ;          /* li contient maintenant : 3, 6, 4, 5, 2
*/
```

b) Suppressions des éléments répondant à une condition

remove_if (prédicat) /* supprime tous les éléments répondant au prédicat */

Cette fonction supprime tous les éléments pour lesquels le prédicat unaire fourni en argument est vrai. La notion de prédicat a été abordée dans le paragraphe 6 du chapitre XVIII. Voici un exemple utilisant le prédicat pair défini ainsi :

```
bool pair(int n)    /* ne pas oublier :   #include <functional>
{ return (n%2) ;
}

int t[] = {1, 6, 3, 9, 11, 18, 5 } ;
list<int> li(t, t+7) ;   /* li contient :            1, 6, 3, 9, 11, 18, 5 */
li.remove_if(pair) ;     /* li contient maintenant : 1, 3, 9, 11, 5        */
```

Remarques

1) La fonction membre remove ne fournit aucun résultat, de sorte qu'il n'est pas possible de savoir s'il existait des éléments répondant aux conditions spécifiées. Il est toujours possible de recourir auparavant à un algorithme tel que count pour obtenir cette information.

2) Il existe une fonction membre unique dont la vocation est également la suppression d'éléments ; cependant, nous vous la présenterons dans le paragraphe suivant, consacré à la fonction de tri (sort) car elle est souvent utilisée conjointement avec la fonction unique.

4.3 Opérations globales

En plus des possibilités générales offertes par l'affectation et la fonction membre assign, décrites au paragraphe 1.2, la classe list en offre d'autres, assez originales : tri de ses éléments avec suppression éventuelle des occurrences multiples, fusion de deux listes préalablement ordonnées, transfert de tout ou partie d'une liste dans une autre liste de même type.

a) Tri d'une liste

Il existe des algorithmes de tri des éléments d'un conteneur, mais la plupart nécessitent des itérateurs à accès direct. En fait, la classe list dispose de sa propre fonction sort, écrite spécifiquement pour ce type de conteneur et relativement efficace, puisqu'en O (Log N).

Comme tout ce qui touche à l'ordonnancement d'un conteneur, la fonction sort s'appuie sur une relation d'ordre faible strict, telle qu'elle a été présentée dans le chapitre précédent. On pourra utiliser par défaut l'opérateur <, y compris pour un type classe, pour peu que cette dernière l'ait convenablement défini. On aura la possibilité, dans tous les cas, d'imposer une relation de son choix par le biais d'un prédicat binaire prédéfini ou non.

sort () /* trie la liste concernée, en s'appuyant sur l'opérateur < */

```
    list<int> li(...) ;     /* on suppose que li contient : 1, 6, 3, 9, 11, 18,
5 */
    li.sort () ;            /* maintenenant li contient :   1, 3, 5, 6,  9, 11,
18 */
```

sort (prédicat) /* trie la liste concernée, en s'appuyant sur le prédicat binaire prédicat */

```
    list<int> li(...) ;      /* on suppose que li contient : 1, 6, 3, 9, 11, 18,
5 */
    li.sort(greater<int>) ; /* maintenenant li contient :   18, 11, 9, 6, 5, 3,
1 */
```

b) Suppression des éléments en double

La fonction unique permet d'éliminer les éléments en double, à condition de la faire porter sur une liste préalablement triée. Dans le cas contraire, elle peut fonctionner mais, alors, elle se contente de remplacer par un seul élément, les séquences de valeurs consécutives identiques, ce qui signifie que, en définitive, la liste pourra encore contenir des valeurs identiques, mais non consécutives.

Comme on peut s'y attendre, cette fonction se fonde par défaut sur l'opérateur == pour décider de l'égalité de deux éléments, cet opérateur devant bien sûr être défini convenablement en cas d'éléments de type classe. Mais on pourra aussi, dans tous les cas, imposer une relation de comparaison de son choix, par le biais d'un prédicat binaire, prédéfini ou non.

On notera bien que si l'on applique unique à une liste triée d'éléments de type classe, il sera préférable d'assurer la compatibilité entre la relation d'ordre utilisée pour le tri (même s'il s'agit de l'opérateur <) et le prédicat binaire d'égalité (même s'il s'agit de ==). Plus précisément, pour obtenir un fonctionnement logique de l'algorithme, il faudra que les classes d'équivalence induites par la relation == soient les mêmes que celles qui sont induites par la relation d'ordre du tri :

unique() /* ne conserve que le premier élément de valeurs consécutives égales (==) */

unique (prédicat) /* ne conserve que le premier élément de valeurs consécutives */
/* satisfaisant au prédicat binaire prédicat */

Voici un exemple qui montre clairement la différence d'effet obtenu, suivant que la liste est triée ou non.

```
int t[] = {1, 6, 6, 4, 6, 5, 5, 4, 2 } ;
list<int> li1(t, t+9) ;     /* li1  contient :            1 6 6 4 6 5 5 4 2 */
list<int> li2=li1 ;         /* li2  contient :            1 6 6 4 6 5 5 4 2 */
li1.unique() ;              /* li1 contient maintenant : 1 6 4 6 5 4 2     */
li2.sort() ;                /* li2 contient maintenant : 1 2 4 4 5 5 6 6 6 */
li2.unique()                /* li2 contient maintenant : 1 2 4 5 6         */
```

c) Fusion de deux listes

Bien qu'il existe un algorithme général de fusion pouvant s'appliquer à deux conteneurs convenablement triés, la classe list dispose d'une fonction membre spécialisée généralement légèrement plus performante, même si, dans les deux cas, l'efficacité est en O(N1+N2), N1 et N2 désignant le nombre d'éléments de chacune des listes concernées.

La fonction membre merge permet de venir fusionner une autre liste de même type avec la liste concernée. La liste fusionnée est vidée de son contenu. Comme on peut s'y attendre, la fonction merge s'appuie, comme sort, sur une relation d'ordre faible strict ; par défaut, il s'agira de l'opérateur <.

merge (liste) /* fusionne liste avec la liste concernée, en s'appuyant sur l'opérateur < */
/* à la fin : liste est vide */

merge (liste, prédicat) /* fusionne liste avec la liste concernée, en s'appuyant sur le */
/* prédicat binaire prédicat */

On notera qu'en théorie, aucune contrainte ne pèse sur l'ordonnancement des deux listes concernées. Cependant, la fonction merge suppose que les deux listes sont triées suivant la même relation d'ordre que celle qui est utilisée par la fusion. Voici un premier exemple, dans lequel nous avons préalablement trié les deux listes :

```
int t1[] = {1, 6, 3, 9, 11, 18, 5 } ;
int t2[] = {12, 4, 9, 8} ;
list<int> li1(t1, t1+7) ;
list<int> li2(t2, t2+4) ;
li1.sort() ;        /* li1 contient :              1 3 5 6 9 11 18
*/
li2.sort() ;        /* li2 contient :              4 8 9 12
*/
li1.merge(li2) ;    /* li1 contient maintenant :   1 3 4 5 6 8 9 9 11 12 18
*/
                    /* et li2 est vide
*/
```

A simple titre indicatif, voici le même exemple, sans tri préalable des deux listes :

```
  int t1[] = {1, 6, 3, 9, 11, 18, 5 } ;
  int t2[] = {12, 4, 9, 8} ;
  list<int> li1(t1, t1+7) ; /* li1 contient :     1 6 3 9 11 18 5
*/
  list<int> li2(t2, t2+4) ; /* li2 contient :     12 4 9 8
*/
  li1.merge(li2) ;          /* li1 contient maintenant :  1 6 3 9 11 12 4 9 8
18 5 */
                            /* et li2 est vide
*/
```

d) Transfert d'une partie de liste dans une autre

La fonction splice permet de déplacer des éléments d'une autre liste dans la liste concernée. On notera bien, comme avec merge, les éléments déplacés sont supprimés de la liste d'origine et pas seulement copiés.

splice (position, liste_or) /* déplace les éléments de liste_or à l'emplacement position */

```
  char t1[] = {"xyz"}, t2[] = {"abcdef"} ;
  list<char> li1(t1, t1+3) ;  /* li1 contient :     x y z             */
  list<char> li2(t2, t2+6) ;  /* li2 contient :     a b c d e f       */
  list<char>::iterator il ;
  il = li1.begin() ; il++ ;   /* il pointe sur le deuxième élément de li1 */
  li1.splice(il, li2) ;       /* li1 contient :     x a b c d e f y z     */
                              /* li2 est vide                             */
```

splice (position, liste_or, position_or)
/* déplace l'élément de liste_or pointé par position_or à l'emplacement position */

```
  char t1[] = {"xyz"}, t2[] = {"abcdef"} ;
  list<char> li1(t1, t1+3) ;     /* li1 contient :     x y z             */
  list<char> li2(t2, t2+6) ;     /* li2 contient :     a b c d e f       */
  list<char>::iterator il1=li1.begin() ;
  list<char>::iterator il2=li2.end() ; il2-- ;  /* pointe sur avant dernier */
  li1.splice(il1, li2, il2) ;    /* li1 contient :     f x y z             */
                                 /* li2 contient :     a b c d e           */
```

splice (position, liste_or, debut_or, fin_or)
/* déplace l'intervalle [debut_or, fin_or) de la liste liste_or à l'emplacement position */

```
  char t1[] = {"xyz"}, t2[] = {"abcdef"} ;
  list<char> li1(t1, t1+3) ;              /* li1 contient :     x y z
*/
  list<char> li2(t2, t2+6) ;              /* li2 contient :     a b c d e f
*/
  list<char>::iterator il1=li1.begin() ;
  list<char>::iterator il2=li2.begin() ; il2++ ;
  li1.splice(il1, li2, il2, li2.end()) ;  /* li1 contient :     b c d e f x y z
*/
                                          /* li2 contient :     a
*/
```

4.4 Gestion de l'emplacement mémoire

La norme n'impose pas explicitement la manière de gérer les emplacements mémoire alloués à une liste, pas plus qu'elle ne le fait pour les autres conteneurs. Cependant, elle impose à la fois des contraintes d'efficacité

et des règles d'invalidation des itérateurs et des références sur des éléments d'une liste. Notamment, elle précise qu'en cas d'insertions ou de suppressions d'éléments dans une liste, seuls les itérateurs ou références concernant les éléments insérés ou supprimés deviennent invalides. Cela signifie donc que les autres n'ont pas dû changer de place. Ainsi, indirectement, la norme impose que chaque élément dispose de son propre bloc de mémoire.

Dans ces conditions, si le conteneur list dispose toujours des fonctions d'information size() et max_size(), on n'y retrouve en revanche aucune fonction permettant d'agir sur les allocations, et en particulier capacity et reserve.

4.5 Exemple

Voici un exemple complet de programme illustrant bon nombre des fonctionnalités de la classe list que nous avons examinées dans ce paragraphe, ainsi que dans le paragraphe 1.

```
#include <iostream.h>
#include <list>
using namespace std ;
main()
{ void affiche(list<char>) ;
  char mot[] = {"anticonstitutionnellement"} ;
  list<char> lc1 (mot, mot+sizeof(mot)-1) ;
  list<char> lc2 ;
  cout << "lc1 init   : " ; affiche(lc1) ;
  cout << "lc2 init   : " ; affiche(lc2) ;

  list<char>::iterator il1, il2 ;
  il2 = lc2.begin() ;
  for (il1=lc1.begin() ; il1!=lc1.end() ; il1++)
   if (*il1!='t') lc2.push_back(*il1) ; /* equivaut a : lc2=lc1 ;
lc2.remove('t'); */
  cout << "lc2 apres  : " ; affiche(lc2) ;

  lc1.remove('t') ;
  cout << "lc1 remove : " ; affiche(lc1) ;
  if (lc1==lc2) cout << "les deux listes sont egales\n" ;

  lc1.sort() ;
  cout << "lc1 sort   : " ; affiche(lc1) ;
  lc1.unique() ;
  cout << "lc1 unique : " ; affiche(lc1) ;
}
void affiche(list<char> lc)
{ list<char>::iterator il ;
  for (il=lc.begin() ; il!=lc.end() ; il++) cout << (*il) << " " ;
  cout << "\n" ;
}
```

```
lc1 init   : a n t i c o n s t i t u t i o n n e l l e m e n t
lc2 init   :
lc2 apres  : a n i c o n s i u i o n n e l l e m e n
```

```
lc1 remove : a n i c o n s i u i o n n e l l e m e n
les deux listes sont egales
lc1 sort   : a c e e e i i i l l m n n n n n o o s u
lc1 unique : a c e i l m n o s u
```

Exemple d'utilisation de la classe list

5. LES ADAPTATEURS DE CONTENEUR : QUEUE, STACK ET PRIORITY_QUEUE

La bibliothèque standard dispose de trois patrons particuliers stack, queue et priority_queue, dits adaptateurs de conteneurs. Il s'agit de classes patrons construites sur un conteneur d'un type donné qui en modifient l'interface, à la fois en la restreignant et en l'adaptant à des fonctionnalités données. Ils disposent tous d'un constructeur sans argument.

5.1 L'adaptateur stack

Le patron stack est destiné à la gestion de piles de type LIFO (Last In, First Out) ; il peut être construit à partir de l'un des trois conteneurs séquentiels vector, deque ou list, comme dans ces déclarations :

```
stack <int, vector<int> > s1 ;  /* pile de int, utilisant un conteneur vector
*/
stack <int, deque<int> >  s2 ;  /* pile de int, utilisant un conteneur deque
*/
stack <int, list<int> >   s3 ;  /* pile de int, utilisant un conteneur list
*/
```

Dans un tel conteneur, on ne peut qu'introduire (push) des informations qu'on empile les unes sur les autres et qu'on recueille, à raison d'une seule à la fois, en extrayant la dernière introduite. On y trouve uniquement les fonctions membre suivantes :

- empty() : fournit true si la pile est vide,
- size() : fournit le nombre d'éléments de la pile,
- top() : accès à l'information située au sommet de la pile qu'on peut connaître ou modifier (sans la supprimer),
- push (valeur) : place valeur sur la pile,
- pop() : supprime l'élément situé au sommet en le supprimant de la pile.

Voici un petit exemple de programme utilisant une pile :

```
#include <iostream.h>
#include <stack>
#include <vector>
using namespace std ;
main()
{
  int i ;
```

```
  stack<int, vector<int> > q ;
  cout << "taille initiale  : " << q.size() << "\n" ;
  for (i=0 ; i<10 ; i++) q.push(i*i) ;
  cout << "taille apres for  : " << q.size() << "\n"  ;
  cout << "sommet de la pile : " << q.top()  << "\n" ;
  q.top() = 99 ;   /* on modifie le sommet de la pile */
  cout << "on depile : " ;
  for (i=0 ; i<10 ; i++)  { cout << q.top() << " " ; q.pop() ; }
}
```

```
taille initiale  : 0
taille apres for  : 10
sommet de la pile : 81
on depile : 99 64 49 36 25 16 9 4 1 0
```

Exemple d'utilisation de l'adaptateur de conteneur stack

5.2 L'adaptateur queue

Le patron queue est destiné à la gestion de files d'attentes, dites aussi queues, ou encore piles de type FIFO (First In, First Out). On y place des informations qu'on introduit en fin et qu'on recueille en tête, dans l'ordre inverse de leur introduction. Un tel conteneur peut être construit à partir de l'un des deux conteneurs séquentiels deque ou list (le conteneur vector ne serait pas approprié puisqu'il ne dispose pas d'insertions efficaces en début), comme dans ces déclarations :

```
queue <int, deque<int> >  q1 ;  /* queue de int, utilisant un conteneur deque
*/
queue <int, list<int> >   q2 ;  /* queue de int, utilisant un conteneur list
*/
```

On y trouve uniquement les fonctions membre suivantes :

- empty() : fournit true si la queue est vide,
- size() : fournit le nombre d'éléments de la queue,
- front() : accès à l'information située en tête de la queue, qu'on peut ainsi connaître ou modifier, sans la supprimer,
- back() : accès à l'information située en fin de la queue, qu'on peut ainsi connaître ou modifier, sans la supprimer,
- push (valeur) : place valeur dans la queue,
- pop() : fournit l'élément situé en tête de la queue en le supprimant.

Voici un petit exemple de programme utilisant une queue :

```
#include <iostream.h>
#include <queue>
#include <deque>
```

```
using namespace std ;
main()
{ int i ;
  queue<int, deque<int> > q ;
  for (i=0 ; i<10 ; i++) q.push(i*i) ;
  cout << "tete de la queue : " << q.front() << "\n" ;
  cout << "fin de la queue : "  << q.back()  << "\n" ;
  q.front() = 99 ;    /* on modifie la tete de la queue */
  q.back()  = -99 ;   /* on modifie la fin de la queue */
  cout << "on depile la queue : " ;
  for (i=0 ; i<10 ; i++)
  { cout << q.front() << " " ; q.pop() ;
  }
}
```

```
tete de la queue : 0
fin de la queue : 81
on depile la queue : 99 1 4 9 16 25 36 49 64 -99
```

Exemple d'utilisation de l'adaptateur de conteneur stack

5.3 L'adaptateur priority_queue

Un tel conteneur ressemble à une file d'attente, dans laquelle on introduit toujours des éléments en fin ; en revanche, l'emplacement des éléments dans la queue est modifié à chaque introduction, de manière à respecter une certaine priorité définie par une relation d'ordre qu'on peut fournir sous forme d'un prédicat binaire. On parle parfois de file d'attente avec priorités. Un tel conteneur ne peut être construit qu'à partir d'un conteneur deque, comme dans ces déclarations :

```
priority_queue <int, deque<int> > q1 ;
priority_queue <int, deque<int>, greater<int> > q2 ;
```

En revanche, ici, on peut le construire classiquement à partir d'une séquence.

On y trouve uniquement les fonctions membre suivantes :

- empty() : fournit true si la queue est vide ;
- size() : fournit le nombre d'éléments de la queue ;
- push (valeur) : place valeur dans la queue ;
- top() : accès à l'information située en tête de la queue qu'on peut connaître ou, théoriquement modifier (sans la supprimer) ; actuellement, nous recommandons de ne pas utiliser la possibilité de modification qui, dans certaines implémentations, n'assure plus le respect de l'ordre des éléments de la queue ;
- pop() : fournit l'élément situé en tête de la queue en le supprimant.

Voici un petit exemple de programme utilisant une file d'attente avec priorités :

```
#include <iostream.h>
#include <queue>
```

```
#include <deque>

using namespace std ;
main()
{ int i ;
  priority_queue <int, deque<int>, greater<int> > q ;
  q.push (10) ; q.push(5) ; q.push(12) ; q.push(8) ;
  cout << "tete de la queue : " << q.top() << "\n" ;
  cout << "on depile : " ;
  for (i=0 ; i<4 ; i++) { cout << q.top() << " " ; q.pop() ;
                        }
}                       ________________________

tete de la queue : 5
on depile : 5 8 10 12
```

Exemple d'utilisation de l'adaptateur de conteneur priority_queue

XX. LES CONTENEURS ASSOCIATIFS

Comme il a été dit dans le chapitre XVIII, les conteneurs se classent en deux catégories : les conteneurs séquentiels et les conteneurs associatifs. Les conteneurs séquentiels, que nous avons étudiés dans le précédent chapitre, sont ordonnés suivant un ordre imposé explicitement par le programme lui-même ; on accède à un de leurs éléments en tenant compte de cet ordre, que l'on utilise un indice ou un itérateur.

Les conteneurs associatifs ont pour principale vocation de retrouver une information, non plus en fonction de sa place dans le conteneur, mais en fonction de sa valeur ou d'une partie de sa valeur nommée clé. Nous avons déjà cité l'exemple du répertoire téléphonique, dans lequel on retrouve le numéro de téléphone à partir d'une clé formée du nom de la personne concernée. Malgré tout, pour de simples questions d'efficacité, un conteneur associatif se trouve ordonné intrinsèquement en permanence, en se fondant sur une relation (par défaut <) choisie à la construction.

Les deux conteneurs associatifs les plus importants sont map et multimap. Ils correspondent pleinement au concept de conteneur associatif, en associant une clé et une valeur. Mais, alors que map impose l'unicité des clés, autrement dit l'absence de deux éléments ayant la même clé, multimap ne l'impose pas et on pourra y trouver plusieurs éléments de même clé qui apparaîtront alors consécutivement. Si l'on reprend notre exemple de répertoire téléphonique, on peut dire que multimap autorise la présence de plusieurs personnes de même nom (avec des numéros associés différents ou non), tandis que map ne l'autorise pas. Cette distinction permet précisément de redéfinir l'opérateur [ ] sur un conteneur de type map. Par exemple, avec un conteneur nommé annuaire, dans lequel les clés sont des chaînes, on pourra utiliser l'expression annuaire ["Dupont"] pour désigner l'élément correspondant à la clé "Dupont" ; cette possibilité n'existera naturellement plus avec multimap.

Il existe deux autres conteneurs qui correspondent à des cas particuliers de map et multimap, dans le cas où la valeur associée à la clé n'existe plus, ce qui revient à dire que les éléments se limitent à la seule clé. Dans ces conditions, la notion d'association entre une clé et une valeur disparaît et il ne reste plus que la notion d'appartenance. Ces conteneurs se nomment set et multiset et l'on verra qu'effectivement ils permettront de représenter des ensembles au sens mathématique, à condition toutefois de disposer, comme pour tout conteneur associatif, d'une relation d'ordre appropriée sur les éléments, ce qui n'est pas nécessaire en mathématiques ; en outre multiset autorisera la présence de plusieurs éléments identiques, ce qui n'est manifestement pas le cas d'un ensemble usuel.

1. LE CONTENEUR MAP

Le conteneur map est donc formé d'éléments composés de deux parties : une clé et une valeur. Pour représenter de tels éléments, il existe un patron de classe approprié, nommé pair, paramétré par le type de la clé et par celui de la valeur. Un conteneur map permet d'accéder rapidement à la valeur associée à une clé en utilisant l'opérateur [] ; l'efficacité de l'opération est en O(N Log N). Comme un tel conteneur est ordonné en permanence, cela suppose le recours à une relation d'ordre qui, comme à l'accoutumée, doit posséder les propriétés d'une relation d'ordre faible strict, telles qu'elles ont été présentées au chapitre XVIII.

Comme la notion de tableau associatif est moins connue que celle de tableau, de vecteur ou même que celle de liste, nous commencerons par un exemple introductif d'utilisation d'un conteneur de type map avant d'en étudier les propriétés en détail.

1.1 Exemple introductif

Une déclaration telle que :

```
map<char, int> m ;
```

crée un conteneur de type map, dans lequel les clés sont de type char et les valeurs associées de type int. Pour l'instant, ce conteneur est vide : m.size() vaut 0.

Une instruction telle que :

```
m['S'] = 5 ;
```

insère, dans le conteneur m, un élément formé de l'association de la clé 'S' et de la valeur 5. On voit déjà là une différence fondamentale entre un vecteur et un conteneur de type map : dans un vecteur, on ne peut accéder par l'opérateur [ ] qu'aux éléments existants et, en aucun cas, en insérer de nouveaux.

Qui plus est, si l'on cherche à utiliser une valeur associée à une clé inexistante, comme dans :

```
cout << "valeur associée a la clé 'X' : ", m['X'] ;
```

le simple fait de chercher à consulter m['X'] créera l'élément correspondant, en initialisant la valeur associée à 0.

Pour afficher tous les éléments d'un map tel que m, on pourra le parcourir avec un itérateur bidirectionnel classique iterator fourni par la classe map. Ceci n'est possible que parce que, comme nous l'avons dit à plusieurs reprises, les conteneurs associatifs sont ordonnés intrinsèquement. On pourra classiquement parcourir tous les éléments de m par l'un des deux schémas suivants :

```
map<char, int> ::iterator im ;    /* itérateur sur un map<char,int> */
  .....
for (im=m.begin() ; im!=m.end() ; im++)   /* im parcourt tout le map m */
  { /* ici *im désigne l'élément courant de m */
  }

map<char, int> ::reverse_iterator im ;   /* itérateur inverse sur un
map<char,int> */
  .....
for (im=m.rbegin() ; im!=m.rend() ; im++)   /* im parcourt tout le map m */
```

```
    { /* ici *im désigne l'élément courant de m */
    }
```

Cependant, on constate qu'une petite difficulté apparaît : *im désigne bien l'élément courant de m, mais, la plupart du temps, on aura besoin d'accéder séparément à la clé et à la valeur correspondante. En fait, les éléments d'un conteneur map sont d'un type classe particulier, nommé pair, qui dispose de deux membres publics :

- first correspondant à la clé,
- second correspondant à la valeur associée.

En définitive, voici, par exemple, comment afficher, suivant l'ordre naturel, toutes les valeurs de m sous la forme (clé, valeur) :

```
for (im=m.begin() ; im!=m.end() ; im++)
    cout << "(" << (*im).first << "," << (*im).second << ") " ;
```

Voici un petit programme complet reprenant les différents points que nous venons d'examiner (attention, la position relative de la clé 'c' peut dépendre de l'implémentation) :

```
#include <iostream.h>
#include <map>
using namespace std ;
main()
{ void affiche (map<char, int>) ;
  map<char, int> m ;
  cout << "map initial : " ; affiche(m) ;
  m['S'] = 5 ;   /* la cle S n'existe pas encore, l'element est cree */
  m['C'] = 12 ;  /* idem */
  cout << "map SC      : " ; affiche(m) ;
  cout << "valeur associee a la cle 'S' : " << m['S'] << "\n" ;
  cout << "valeur associee a la cle 'X' : " << m['X'] << "\n" ;
  cout << "map X       : " ; affiche(m) ;
  m['S'] = m['c'] ;  /* on a utilise m['c'] au lieu de m['C'] ; */
                     /* la cle 'c' est creee                    */
  cout << "map final   : " ; affiche(m) ;
}
void affiche (map<char, int> m)
{ map<char, int> ::iterator im ;
  for (im=m.begin() ; im!=m.end() ; im++)
    cout << "(" << (*im).first << "," << (*im).second << ") " ;
  cout << "\n" ;
}
```

```
map initial :
map SC      : (C,12) (S,5)
valeur associee a la cle 'S' : 5
valeur associee a la cle 'X' : 0
map X       : (C,12) (S,5) (X,0)
map final   : (C,12) (S,0) (X,0) (c,0)
```

1.2 Le patron de classes pair

Comme nous venons de le voir, il existe un patron de classe pair, comportant deux paramètres de type et permettant de regrouper dans un objet deux valeurs. On y trouve un constructeur à deux arguments :

```
pair <int, float> p(3, 1.25) ;    /* crée une paire formée d'un int de valeur 3
*/
                                  /* et d'un float de valeur 1.25
*/
```

Pour affecter des valeurs données à une telle paire, on peut théoriquement procéder comme dans :

```
p = pair<int,float> (4, 3.35) ;  /* ici, les arguments peuvent être d'un type
*/
                                  /* compatible par affectation avec celui
attendu */
```

Mais les choses sont un peu plus simples si l'on fait appel à une fonction standard make_pair :

```
p = make_pair (4, 3.35f) ;    /* attention : 3.35f car le type des arguments
*/
                              /* sert à instancier la fonction patron make_pair
*/
```

Comme on l'a vu dans notre exemple introductif, la classe pair dispose de deux membres publics nommés first et second. Ainsi, l'instruction précédente pourrait également s'écrire :

```
p.first = 4 ; p.second = 3.35 ;  /* ici 3.35 de type double sera converti en
float */
```

La classe pair dispose des deux opérateurs == et <. Le second correspond à une comparaison lexicographique, c'est-à-dire qu'il applique d'abord < à la clé, puis à la valeur. Bien entendu, dans le cas où l'un des éléments au moins de la paire est de type classe, ces opérateurs doivent être convenablement surdéfinis.

1.3 Construction d'un conteneur de type map

Les possibilités de construction d'un tel conteneur sont beaucoup plus restreintes que pour les conteneurs séquentiels ; elles se limitent à trois possibilités :

- construction d'un conteneur vide (comme dans notre exemple du paragraphe 1.1) ;
- construction à partir d'un autre conteneur de même type ;
- construction à partir d'une séquence.

En outre, il est possible de choisir la relation d'ordre qui sera utilisée pour ordonner intrinsèquement le conteneur. Pour plus de clarté, nous examinerons ce point à part.

a) Constructions utilisant la relation d'ordre par défaut

Construction d'un conteneur vide

On se contente de préciser les types voulus pour la clé et pour la valeur, comme dans ces exemples (on suppose que point est un type classe) :

```
map <int, long>    m1 ;     /* clés de type int,  valeurs associées de type
long   */
map <char, point>  m2 ;     /* clés de type char, valeurs associées de type
point  */
map <string, long> repert ; /* clés de type string, valeurs associées de type
long */
```

Construction à partir d'un autre conteneur de même type

Il s'agit d'un classique constructeur par recopie qui, comme on peut s'y attendre, appelle le constructeur par recopie des éléments concernés lorsqu'il s'agit d'objets.

```
map <int, long> m1 ;
   .....
map <int, long> m2(m1) ;     /* ou encore :  map <int, long> m2 = m1 ;  */
```

Construction à partir d'une séquence

Il s'agit d'une possibilité déjà rencontrée pour les conteneurs séquentiels, avec cependant une différence importante : les éléments concernés doivent être de type pair< type_des_clés, type_des_valeurs> . Par exemple, s'il existe une liste lr, construite ainsi :

```
list<pair<char, long> > lr (...) ;
```

et convenablement remplie, on pourra l'utiliser en partie ou en totalité pour construire :

```
map <char, long> repert (lr.begin(), lr.end() ) ;
```

En pratique, ce type de construction est peu utilisé.

b) Choix de l'ordre intrinsèque du conteneur

Comme on l'a déjà dit, les conteneurs sont intrinsèquement ordonnés en faisant appel à une relation d'ordre faible strict pour ordonner convenablement les clés. Par défaut, on utilise la relation < , qu'il s'agisse de la relation prédéfinie pour les types scalaires ou string, ou d'une surdéfinition de l'opérateur > lorsque les clés sont des objets.

Il est possible d'imposer à un conteneur d'être ordonné en utilisant une autre relation que l'on fournit sous forme d'un prédicat binaire prédéfini (comme less< int>) ou non. Dans ce dernier cas, il est alors nécessaire de fournir un type et non pas un nom de fonction, ce qui signifie qu'il est nécessaire de recourir à une classe fonction (dont nous avons parlé au chapitre XVII). Voici quelques exemples :

```
map <char, long, greater<char> > m1 ;   /* les clés seront ordonnées par
valeurs   */
                                        /* décroissantes - attention > > et non
>> */
map <char, long, greater<char> > m2(m1) ;  /* si m2 n'est pas ordonné par la
même  */
```

```
                                                    /* relation → erreur de compilation
*/

class mon_ordre
{ .....
  public :
   bool operator () (int n, int p) { ..... }  /* ordre faible strict */
} ;
map <int, float, mon_ordre> m_perso ;  /* clés ordonnées par le prédicat
mon_ordre */
                                       /* qui doit être une classe fonction
*/
```

Remarque

Certaines implémentations peuvent ne pas accepter le choix d'une valeur par défaut pour la relation d'ordre des clés. Dans ce cas, il faut toujours préciser less< type> comme troisième argument, type correspondant au type des clés pour instancier convenablement le conteneur. La lourdeur des notations qui en découle peut parfois inciter à recourir à l'instruction typedef.

c) Pour connaître la relation d'ordre utilisée par un conteneur

Les classes map disposent d'une fonction membre key_comp() fournissant la fonction utilisée pour ordonner les clés. Par exemple, avec le conteneur de notre exemple introductif :

```
map<char, int> m ;
```

on peut, certes, comparer deux clés de type char de façon directe, comme dans :

```
if ('a' < 'c') .....
```

mais, on obtiendra le même résultat avec :

```
if m.key_comp() ('a', 'c') .....   /* notez bien key_comp() (....) */
```

Certes, tant que l'on se contente d'ordonner de tels conteneurs en utilisant la relation d'ordre par défaut, ceci ne présente guère d'intérêt ; dans le cas contraire, cela peut éviter d'avoir à se demander, à chaque fois qu'on compare des clés, quelle relation d'ordre a été utilisée lors de la construction.

D'une manière similaire, la classe map dispose d'une fonction membre value_comp() fournissant la fonction utilisable pour comparer deux éléments, toujours selon la valeur des clés. L'intérêt de cette fonction est de permettre de comparer deux éléments (donc, deux paires), suivant l'ordre des clés, sans avoir à en extraire les membres first. On notera bien que, contrairement à key_comp, cette fonction n'est jamais choisie librement, elle est simplement déduite de key_comp. Par exemple, avec :

```
map <char, int> m ;
map <char, int>::iterator im1, im2 ;
```

on pourra comparer les clés relatives aux éléments pointés par im1 et im2 de cette manière :

```
if ( value_comp() (*im1, *im2) ) .....
```

Avec key_comp, il aurait fallu procéder ainsi :

```
if ( key_comp() ( (*im1).first, (*im2).first) ) .....
```

d) Conséquences du choix de l'ordre d'un conteneur

Tant que l'on utilise des clés de type scalaire ou string et qu'on se limite à la relation par défaut (<), aucun problème particulier ne se pose. Il n'en va plus nécessairement de même dans les autres cas.

Par exemple, on dit généralement que, dans un conteneur de type map, les clés sont uniques. En fait, pour être plus précis, il faudrait dire qu'un nouvel élément n'est introduit dans un tel conteneur que s'il n'existe pas d'autre élément possédant une clé équivalente ; l'équivalence étant celle qui est induite par la relation d'ordre, tel qu'il a été expliqué dans le paragraphe 7.2 du chapitre XVIII. Par exemple, considérons un map utilisant comme clé des objets de type point et supposons que la relation < ait été définie dans la classe point en s'appuyant uniquement sur les abscisses des points ; dans ces conditions, les clés correspondant à des points de même abscisse apparaîtront comme équivalentes.

De plus, comme on aura l'occasion de le voir plus loin, la recherche d'un élément de clé donnée se fondera, non pas sur une hypothétique relation d'égalité, mais bel et bien sur la relation d'ordre utilisée pour ordonner le conteneur. Autrement dit, toujours avec notre exemple de points utilisés en guise de clés, on pourra rechercher la clé (1, 9) et trouver la clé (1, 5).

1.4 Accès aux éléments

Comme tout conteneur, map permet théoriquement d'accéder aux éléments existants, soit pour en connaître la valeur, soit pour la modifier. Cependant, par rapport aux conteneurs séquentiels, ces opérations prennent un tour un peu particulier lié à la nature même des conteneurs associatifs. En effet, d'une part, une tentative d'accès à une clé inexistante amène à la création d'un nouvel élément, d'autre part, comme on le verra un peu plus loin, une tentative de modification globale (clé + valeur) d'un élément existant sera fortement déconseillée.

a) Accès par l'opérateur [ ]

Le paragraphe 1 a déjà montré en quoi cet accès par l'opérateur est ambigu puisqu'il peut conduire à la création d'un nouvel élément, dès lors qu'on l'applique à une clé inexistante et cela, aussi bien en consultation qu'en modification. Par exemple :

```
map<char, int> m ;
   .....
m ['S'] = 2 ;  /* si la clé 'S' n'existe pas, on crée l'élément make_pair ('S',
2) */
 /* si la clé existe, on modifie la valeur de l'élément qui ne change pas de
place */
... = m['T'] ; /* si la clé 'T' n'existe pas, on crée l'élément make_pair ('T',
0) */
```

b) Accès par itérateur

Comme on peut s'y attendre et comme on l'a déjà fait dans les exemples précédents, si it est un itérateur valide sur un conteneur de type map, l'expression *it désigne l'élément correspondant ; rappelons qu'il s'agit d'une paire formée de la clé (*it).first et de la valeur associée (*it).second ; en général, d'ailleurs, on sera plutôt amené à s'intéresser à ces deux dernières valeurs (ou à l'une d'entre elles) plutôt qu'à la paire complète *it.

En théorie, il n'est pas interdit de modifier la valeur de l'élément désigné par it ; par exemple, pour un conteneur de type map< char, int> , on pourrait écrire :

```
*it = make_pair ('R', 5) ;     /* remplace théoriquement l'élément désigné par
ip */
                               /*  fortement déconseillé en pratique
*/
```

Mais le rôle exact d'un telle opération n'est actuellement pas totalement spécifié par la norme. Or, certaines ambiguïtés apparaissent. En effet, d'une part, comme une telle opération modifie la valeur de la clé, le nouvel élément risque de ne plus être à sa place ; il devrait donc être déplacé ; d'autre part, que doit-il se passer si la clé 'R' existe déjà ? La seule démarche raisonnable nous semble être de dire qu'une telle modification devrait être équivalente à une destruction de l'élément désigné par it, suivie d'une insertion du nouvel élément. En pratique, ce n'est pas ce que l'on constate dans toutes les implémentations actuelles. Dans ces conditions :

Il est fortement déconseillé de modifier la valeur d'un élément d'un map, par le biais d'un itérateur.

c) Recherche par la fonction membre find

La fonction membre

find (clé)

a un rôle naturel : fournir un itérateur sur un élément ayant une clé donnée (ou une clé équivalente au sens de la relation d'ordre utilisée par le conteneur). Si aucun élément n'est trouvé, cette fonction fournit la valeur end().

Remarque

Attention, la fonction find ne se base pas sur l'opérateur == ; cette remarque est surtout sensible lorsque l'on a affaire à des éléments de type classe, classe dans laquelle on a surdéfini l'opérateur == de manière incompatible avec le prédicat binaire utilisé pour ordonner le conteneur. Les résultats peuvent alors être déconcertants.

1.5 Insertions et suppressions

Comme on peut s'y attendre, le conteneur map offre des possibilités de modifications dynamiques fondées sur des insertions et des suppressions, analogues à celles qui sont offertes par les conteneurs séquentiels. Toutefois, si la notion de suppression d'un élément désigné par un itérateur conserve la même signification, celle d'insertion à un emplacement donné n'a plus guère de raison d'être puisqu'on ne peut plus agir sur la manière dont sont intrinsèquement ordonnés les éléments d'un conteneur associatif. On verra qu'il existe quand même une fonction d'insertion recevant un tel argument mais que ce dernier a en fait un rôle un peu particulier.

En outre, alors qu'une insertion dans un conteneur séquentiel aboutissait toujours, dans le cas d'un conteneur de type map, elle n'aboutit que s'il n'existe pas d'élément de clé équivalente.

D'une manière générale, l'efficacité de ces opérations est en O(Log N). Nous apporterons quelques précisions par la suite pour chacune des opérations.

a) Insertions

La fonction membre insert permet d'insérer :

- un élément de valeur donnée :

insert (élément) /* insère la paire élément */

- les éléments d'un intervalle :

  insert (début, fin) /* insère les paires de la séquence [début, fin) */

On notera bien, dans les deux cas, que les éléments concernés doivent être des paires d'un type approprié.

L'efficacité de la première fonction est en O(Log N) ; celle de la seconde est en O(Log(N+M)), M désignant le nombre d'éléments de l'intervalle. Toutefois, si cet intervalle est trié suivant l'ordre voulu, l'efficacité est en O(M).

Voici quelques exemples :

```
map<int, float> m1, m2 ;
map<int, float>::iterator im1 ;
.....
m1.insert (make_pair(5, 6.25f)) ;      /* tentative d'insertion d'un élément
*/
m1.insert (m2.begin(), m2.end()) ;     /* tentative d'insertion d'une séquence
*/
```

Remarques

1) En toute rigueur, il existe une troisième version de insert, de la forme :

   insert (paire, position)

   L'itérateur position est une suggestion qui est faite pour faciliter la recherche de l'emplacement exact d'insertion. Si la valeur fournie correspond exactement au point d'insertion, on obtient alors une efficacité en O(1), ce qui s'explique par le fait que la fonction n'a besoin que de comparer deux valeurs consécutives.

2) Les deux fonctions d'insertion d'un élément fournissent une valeur de retour qui est une paire de la forme pair(position, indic), dans laquelle le booléen indic précise si l'insertion a eu lieu et position est l'itérateur correspondant ; on notera que son utilisation est assez laborieuse ; voici, par exemple, comment adapter notre précédent exemple dans ce sens :

```
     if(m1.insert(make_pair(5, 6.25f)).second) cout << "insertion effectuée\n"
;
                                          else cout << "élément existant\n" ;
```

   Et encore, ici, nous n'avons pas cherché à placer la valeur de retour dans une variable. Si nous avions voulu le faire, il aurait fallu déclarer une variable, par exemple resul, d'un type pair approprié ; de plus, comme pair ne dispose pas de constructeur par défaut, il aurait fallu préciser des arguments fictifs ; voici une déclaration possible :

```
     pair<map<int,float>::iterator, bool> resul(m1.end(),false) ;
```

   Dans les implémentations qui n'acceptent pas la valeur less< type> par défaut, les choses seraient encore un peu plus complexes et il serait probablement plus sage de recourir à des définitions de types synonymes (typedef) pour alléger quelque peu l'écriture.

b) Suppressions

La fonction erase permet de supprimer :

- un élément de position donnée :

erase (position) /* supprime l'élément désigné par position */

- les éléments d'un intervalle :

  erase (début, fin) /*supprime les paires de l'intervalle [début, fin) */

- l'élément de clé donnée :

  erase (clé) /* supprime les éléments[1] de clé équivalente à clé */

En voici quelques exemples :

```
map<int, float> m ;
map<int, float>::iterator im1, im2 ;
.....
m.erase (5) ;              /* supprime l'élément de clé 5 s'il existe
*/
m.erase (im1) ;            /* supprime l'élément désigné par im1
*/
m.erase (im2, m.end()) ;   /* supprime tous les éléments de celui désigné par
im2 */
                           /*  jusqu'à la fin du conteneur m
*/
```

Enfin, de façon fort classique, la fonction clear() vide le conteneur de tout son contenu.

Remarque

Il peut arriver que l'on souhaite supprimer tous les éléments dont la clé appartient à un intervalle donné. Dans ce cas, on pourra avoir recours aux fonctions lower_bound et upper_bound présentées dans le paragraphe 2.

1.6 Gestion mémoire

Contrairement à ce qui se passe pour certains conteneurs séquentiels, les opérations sur les conteneurs associatifs, donc, en particulier, sur map, n'entraînent jamais d'invalidation des références et des itérateurs, excepté, bien entendu, pour les éléments supprimés qui ne sont plus accessibles après leur destruction.

Toutefois, comme on l'a indiqué dans le paragraphe 1.4, il est théoriquement possible, bien que fortement déconseillé, de modifier globalement un élément de position donnée ; par exemple (iv désignant un itérateur valide sur un conteneur de type map< char, int>) :

```
*iv = make_pair ('S', 45) ;
```

Que la clé 'S' soit présente ou non, on court, outre les risques déjà évoqués, celui que l'itérateur iv devienne invalide.

1.7 Autres possibilités

Les manipulations globales des conteneurs map se limitent à la seule affectation et à la fonction swap permettant d'échanger les contenus de deux conteneurs de même type. Il n'existe pas de fonction assign, ni de possibilités de comparaisons lexicographiques auxquelles il serait difficile de donner une signification ; en effet, d'une part, les éléments sont des paires, d'autre part, un tel conteneur est ordonné intrinsèquement et son organisation évolue en permanence.

---

1 - Pour map, il y en aura un au plus ; pour multimap, on pourra en trouver plusieurs.

En théorie, il existe des fonctions membre lower_bound, upper_bound, equal_range et count qui sont utilisables aussi bien avec des conteneurs de type map qu'avec des conteneurs de type multimap. C'est cependant dans ce dernier cas qu'elles présentent le plus d'intérêt ; elles seront étudiées dans le paragraphe 2.

1.8 Exemple

Voici un exemple complet de programme illustrant les principales fonctionnalités de la classe map que nous venons d'examiner.

```
#include <iostream.h>
#include <map>
using namespace std ;
main()
{ void affiche(map<char, int>) ;
  map<char, int> m ;
  map<char, int>::iterator im ;
  m['c'] = 10 ; m['f'] = 20 ; m['x'] = 30 ; m['p'] = 40 ;
  cout << "map initial          : " ; affiche(m) ;
  im = m.find ('f') ;              /* ici, on ne verifie pas que im est !=
m.end() */
  cout << "cle 'f' avant insert : " << (*im).first << "\n" ;
  m.insert (make_pair('a', 5)) ;   /* on insere un element avant 'f' */
  m.insert (make_pair('t', 7)) ;   /* et un element apres 'f'        */
  cout << "map apres insert     : " ; affiche(m) ;
  cout << "cle 'f' apres insert : " << (*im).first << "\n" ; /* im → 'f' */
  m.erase('c') ;
  cout << "map apres erase 'c'  : " ; affiche(m) ;
  im = m.find('p') ; if (im != m.end()) m.erase(im, m.end()) ;
  cout << "map apres erase int  : " ; affiche(m) ;
}
void affiche(map<char, int> m)
{ map<char, int>::iterator im ;
  for (im=m.begin() ; im!=m.end() ; im++)
    cout << "(" << (*im).first << "," << (*im).second << ") " ;
  cout << "\n" ;
}
```

```
map initial          : (c,10) (f,20) (p,40) (x,30)
cle 'f' avant insert : f
map apres insert     : (a,5) (c,10) (f,20) (p,40) (t,7) (x,30)
cle 'f' apres insert : f
map apres erase 'c'  : (a,5) (f,20) (p,40) (t,7) (x,30)
map apres erase int  : (a,5) (f,20)
```

Exemple d'utilisation de la classe map

2. LE CONTENEUR MULTIMAP

2.1 Présentation générale

Comme nous l'avons déjà dit, dans un conteneur de type multimap, une même clé peut apparaître plusieurs fois ou, plus généralement, on peut trouver plusieurs clés équivalentes. Bien entendu, les éléments correspondants apparaissent alors consécutifs. Comme on peut s'y attendre, l'opérateur [ ] n'est plus applicable à un tel conteneur, compte tenu de l'ambiguïté qu'induirait la non-unicité des clés. Hormis cette restriction, les possibilités des conteneurs map se généralisent sans difficultés aux conteneurs multimap qui possèdent les mêmes fonctions membre, avec quelques nuances qui vont de soi :

- s'il existe plusieurs clés équivalentes, la fonction membre find fournit un itérateur sur un des éléments ayant la clé voulue ; attention, on ne précise pas qu'il s'agit du premier ; celui-ci peut cependant être connu en recourant à la fonction lower_bound examinée un peu plus loin ;
- la fonction membre erase (clé) peut supprimer plusieurs éléments tandis qu'avec un conteneur map, elle n'en supprimait qu'un seul au maximum.

D'autre part, comme nous l'avons déjà fait remarquer, un certain nombre de fonctions membre de la classe map, prennent tout leur intérêt lorsqu'on les applique à un conteneur multimap. On peut, en effet :

- connaître le nombre d'éléments ayant une clé équivalente à une clé donnée, à l'aide de count (clé) ;
- obtenir des informations concernant l'intervalle d'éléments ayant une clé équivalente à une clé donnée, à savoir :

```
lower_bound (clé)    /* fournit un itérateur sur le premier élément ayant une clé */
                     /* équivalente à clé                                          */
upper__bound (clé)   /* fournit un itérateur sur le dernier élément ayant une clé  */
                     /* équivalente à clé                                          */
equal_range (clé)    /* fournit une paire formée des valeurs des deux itérateurs   */
                     /* précédents, lower_bound (clé) et upper_bound (clé)         */
```

On notera qu'on a la relation :

m.equal.range(clé) = make_pair (m.lower_bound (clé), m.upper_bound (clé))

Voici un petit exemple :

```
multimap<char, int> m ;
   .....
m.erase(m.lower_bound('c'), m.upper_bound('c')); /* équivalent tout simplement
à : */
                                                 /*  erase('c') ;
*/
m.erase(m.lower_bound('e'), m.upper_bound('g')); /* supprime toutes les clés
*/
                                                 /* allant de 'e' à 'g'
*/
                                                 /* aucun équivalent simple
*/
```

Remarque

Le deuxième appel de erase de notre précédent exemple peut présenter un intérêt dans le cas d'un conteneur de type map ; en effet, malgré l'unicité des clés dans ce cas, il n'est pas certain qu'un appel tel que :

```
m.erase (m.find('e'), m.find('g')) ;
```

convienne puisqu'on court le risque que l'une au moins des clés 'e' ou 'g' n'existe pas.

2.2 Exemple

Voici un exemple complet de programme illustrant les principales fonctionnalités de la classe multimap que nous venons d'examiner :

```
#include <iostream.h>
#include <map>
using namespace std ;
main()
{
  void affiche(multimap<char, int>) ;
  multimap<char, int> m, m_bis ;
  multimap<char, int>::iterator im ;
  m.insert(make_pair('c', 10)) ;  m.insert(make_pair('f', 20)) ;
  m.insert(make_pair('x', 30)) ;  m.insert(make_pair('p', 40)) ;
  m.insert(make_pair('y', 40)) ;  m.insert(make_pair('p', 35)) ;
  cout << "map initial                : " ; affiche(m) ;
  m.insert(make_pair('f', 25)) ;  m.insert(make_pair('f', 20)) ;
  m.insert(make_pair('x', 2)) ;
  cout << "map avec fff et xx         : " ; affiche(m) ;
  im=m.find('x') ; /* on ne verifie pas que im != m.end() */
  m_bis = m ;      /* on fait une copie de m dans m_bis   */
  m.erase(im) ;
  cout << "map apres erase(find('x')) : " ; affiche(m) ;
  m.erase('f') ;
  cout << "map apres erase('f')       : " ; affiche(m) ;
  m.swap(m_bis) ;
  cout << "map apres swap             : " ; affiche(m) ;
  cout << "il y a " << m.count('f') << " fois la cle 'f'\n" ;
  m.erase(m.upper_bound('f')) ;  /* supprime derniere cle 'f' - ici pas
de test*/
  cout << "map apres erase (u_b('f')) : " ; affiche(m) ;
  m.erase(m.lower_bound('f')) ;
  cout << "map apres erase (l_b('f')) : " ; affiche(m) ;
  m.erase(m.upper_bound('g')) ;
  cout << "map apres erase (u_b('g')) : " ; affiche(m) ;
  m.erase(m.lower_bound('g')) ;
  cout << "map apres erase (l_b('d')) : " ; affiche(m) ;
  m.erase(m.lower_bound('d'), m.upper_bound('x')) ;
  cout << "map apres erase (l_b('d'), u_b('x')) : " ; affiche(m) ;
}
void affiche(multimap<char, int> m)
{ map<char, int>::iterator im ;
  for (im=m.begin() ; im!=m.end() ; im++)
    cout << "(" << (*im).first << "," << (*im).second << ")" ;
  cout << "\n" ;
}
```

```
map initial                : (c,10)(f,20)(p,40)(p,35)(x,30)(y,40)
map avec fff et xx         :
(c,10)(f,20)(f,25)(f,20)(p,40)(p,35)(x,30)(x,2)(y,40)
map apres erase(find('x')) :
(c,10)(f,20)(f,25)(f,20)(p,40)(p,35)(x,2)(y,40)
map apres erase('f')       : (c,10)(p,40)(p,35)(x,2)(y,40)
map apres swap             :
(c,10)(f,20)(f,25)(f,20)(p,40)(p,35)(x,30)(x,2)(y,40)
il y a 3 fois la cle 'f'
map apres erase (u_b('f')) :
(c,10)(f,20)(f,25)(f,20)(p,35)(x,30)(x,2)(y,40)
map apres erase (l_b('f')) : (c,10)(f,25)(f,20)(p,35)(x,30)(x,2)(y,40)
map apres erase (u_b('g')) : (c,10)(f,25)(f,20)(x,30)(x,2)(y,40)
map apres erase (l_b('d')) : (c,10)(f,25)(f,20)(x,2)(y,40)
map apres erase (l_b('d'), u_b('x')) : (c,10)(y,40)
```

Exemple d'utilisation de multimap

3. LE CONTENEUR SET

3.1 Présentation générale

Comme il a été dit en introduction, le conteneur set est un cas particulier du conteneur map, dans lequel aucune valeur n'est associée à la clé. Les éléments d'un conteneur set ne sont donc plus des paires, ce qui en facilite naturellement la manipulation. Une autre différence entre les conteneurs set et les conteneurs map est qu'un élément d'un conteneur set est une constante ; on ne peut pas en modifier la valeur :

```
set<int> e(...)           /* ensemble d'entiers                  */
set<int>::iterator ie ;   /* itérateur sur un ensemble d'entiers */
     .....
cout << *ie ;             /* correct  */
*ie = ... ;               /* interdit */
```

En dehors de cette contrainte, les possibilités d'un conteneur set se déduisent tout naturellement de celles d'un conteneur map, aussi bien pour sa construction que pour l'insertion ou la suppression d'éléments qui, quant à elle, reste toujours possible, aussi bien à partir d'une position que d'une valeur.

3.2 Exemple

Voici un exemple complet de programme illustrant les principales fonctionnalités de la classe set (attention, le caractère "espace" n'est pas très visible dans les résultats !) :

```
#include <iostream.h>
#include <set>
#include <string>
using namespace std ;
```

```
main()
{ char t[] = "je me figure ce zouave qui joue du xylophone" ;
  char v[] = "aeiouy" ;
  void affiche (set<char> ) ;
  set<char> let(t, t+sizeof(t)-1), let_bis ;
  set<char> > voy(v, v+sizeof(v)-1) ;

  cout << "lettres presentes         : " ; affiche (let) ;
  cout << "il y a " << let.size() << " lettres differentes\n" ;
  if (let.count('z'))  cout << "la lettre z est presente\n" ;
  if (!let.count('b')) cout << "la lettre b n'est pas presente\n" ;

  let_bis = let ;
  set<char, less<char> >:: iterator iv ;
  for (iv=voy.begin() ; iv!=voy.end() ; iv++)
    let.erase(*iv) ;
  cout << "lettres sans voyelles     : " ; affiche (let) ;
  let.insert(voy.begin(), voy.end()) ;
  cout << "lettres + toutes voyelles : " ; affiche (let) ;
}
void affiche (set<char> e )
{ set<char>::iterator ie ;
  for (ie=e.begin() ; ie!=e.end() ; ie++)
    cout << *ie << " " ;
  cout << "\n" ;
}
```

```
lettres presentes         :   a c d e f g h i j l m n o p q r u v x y z
il y a 22 lettres differentes
la lettre z est presente
la lettre b n'est pas presente
lettres sans voyelles     :   c d f g h j l m n p q r v x z
lettres + toutes voyelles :   a c d e f g h i j l m n o p q r u v x y z
```

Exemple d'utilisation du conteneur set

3.3 Le conteneur set et l'ensemble mathématique

Un conteneur de type set est obligatoirement ordonné, tandis qu'un ensemble mathématique ne l'est pas nécessairement. Il faudra tenir compte de cette remarque dès que l'on sera amené à créer un ensemble d'objets puisqu'il faudra alors munir la classe correspondante d'une relation d'ordre faible strict. En outre, il ne faudra pas perdre de vue que c'est cette relation qui sera utilisée pour définir l'égalité de deux éléments et non une éventuelle surdéfinition de l'opérateur ==.

Par ailleurs, dans la classe set, il n'existe pas de fonction membre permettant de réaliser les opérations ensemblistes classiques (intersection, réunion...). Cependant, nous verrons, dans le chapitre XXI, qu'il existe des algorithmes généraux, utilisables avec n'importe quelle séquence ordonnée. Leur application au cas particulier des ensembles permettra de réaliser les opérations en question.

4. LE CONTENEUR MULTISET

De même que le conteneur multimap est un conteneur map, dans lequel on autorise plusieurs clés équivalentes, le conteneur multiset est un conteneur set, dans lequel on autorise plusieurs éléments équivalents à apparaître. Bien entendu, cette notion n'a alors plus grand-chose à voir avec la notion mathématique correspondante. Cela n'empêchera pas les algorithmes généraux d'intersection ou de réunion, évoqués ci-dessus, de fonctionner encore dans le cas des conteneurs multiset.

Voici un exemple complet de programme illustrant les principales fonctionnalités de la classe multiset (attention, le caractère "espace" n'est pas très visible dans les résultats !) :

```
#include <iostream.h>
#include <set>
using namespace std ;
main()
{ char t[] = "je me figure ce zouave qui joue du xylophone" ;
  char v[] = "aeiouy" ;
  void affiche (multiset<char> ) ;
  multiset<char> let(t, t+sizeof(t)-1), let_bis ;
  multiset<char> > voy(v, v+sizeof(v)-1) ;
  cout << "lettres presentes    : " ; affiche (let) ;
  cout << "il y a " << let.size() << " lettres en tout\n" ;
  cout << "la lettre e est presente " << let.count('e') << " fois\n" ;
  cout << "la lettre b est presente " << let.count('b') << " fois\n" ;
  let_bis = let ;
  multiset<char>:: iterator iv ;
  for (iv=voy.begin() ; iv!=voy.end() ; iv++)
    let.erase(*iv) ;
  cout << "lettres sans voyelles : " ; affiche (let) ;
}
void affiche (multiset<char> e )
{ multiset<char>::iterator ie ;
  for (ie=e.begin() ; ie!=e.end() ; ie++)
    cout << *ie ;
  cout << "\n" ;
}
```

```
lettres presentes    :         acdeeeeeeefghiijjlmnoooopqruuuuuvxyz
il y a 44 lettres en tout
la lettre e est presente 7 fois
la lettre b est presente 0 fois
lettres sans voyelles :         cdfghjjlmnpqrvxz
```

Exemple d'utilisation du conteneur multiset

5. CONTENEURS ASSOCIATIFS ET ALGORITHMES

Il est généralement difficile d'appliquer certains algorithmes généraux aux conteneurs associatifs. Il y a plusieurs raisons à cela.

Tout d'abord, un conteneur de type map ou multimap est formé d'éléments de pair, qui se prêtent assez difficilement aux algorithmes usuels. Par exemple, une recherche par find devrait se faire sur la paire (clé, valeur), ce qui ne présente généralement guère d'intérêt ; on préférera utiliser la fonction membre find travaillant sur une clé donnée.

De même, vouloir trier un conteneur associatif déjà ordonné de façon intrinsèque n'est guère réaliste : soit on cherche à trier suivant l'ordre interne, ce qui n'a aucun intérêt, soit on cherche à trier suivant un autre ordre, et alors apparaissent des conflits entre les deux ordres.

Néanmoins, il reste possible d'appliquer tout algorithme qui ne modifie pas les valeurs du conteneur.

D'une manière générale, dans le chapitre XXI consacré aux algorithmes, nous indiquerons ceux qui sont utilisables avec des conteneurs associatifs.

XXI. LES ALGORITHMES STANDARD

La notion d'algorithme a déjà été présentée dans le chapitre XVIII, et nous avons eu l'occasion d'en utiliser quelques-uns dans certains de nos précédents exemples. Le présent chapitre expose les différentes possibilités offertes par les algorithmes de la bibliothèque standard. Auparavant, il présente ou rappelle un certain nombre de notions générales qui interviennent dans leur utilisation, en particulier : les catégories d'itérateur, la notion de séquence, les itérateurs de flot et les itérateurs d'insertion.

On notera bien que ce chapitre vise avant tout à faire comprendre le rôle des différents algorithmes et à illustrer les plus importants par des exemples de programmes. On trouvera dans l'annexe C, une référence complète du rôle précis, de l'efficacité et de la syntaxe exacte de l'appel de chacun des algorithmes existants.

1. NOTIONS GENERALES

1.1 Algorithmes et itérateurs

Les algorithmes standard se présentent sous forme de patrons de fonctions. Leur code est écrit, sans connaissance précise des éléments qu'ils seront amenés à manipuler. Cependant, cette manipulation ne se fait jamais directement, mais toujours par l'intermédiaire d'un itérateur qui, quant à lui, possède un type donné, à partir duquel se déduit le type des éléments effectivement manipulés. Par exemple, lorsqu'un algorithme contient une instruction de la forme :

```
*it = ...
```

le code source du programme ne connaît effectivement pas le type de l'élément qui sera ainsi manipulé mais ce type sera parfaitement défini à la compilation, lors de l'instanciation de la fonction patron correspondant à l'algorithme en question.

1.2 Les catégories d'itérateurs

Jusqu'ici, nous avons surtout manipulé des éléments de conteneurs et les itérateurs associés qui se répartissaient alors en trois catégories : unidirectionnel, bidirectionnel et à accès direct. En fait, il existe deux autres catégories d'itérateurs, disposant de propriétés plus restrictives que les itérateurs unidirectionnels ; il s'agit des itérateurs en entrée et des itérateurs en sortie. Bien qu'ils ne soient fournis par aucun des conteneurs, ils présentent un intérêt au niveau des itérateurs de flot qui, comme nous le verrons un peu plus loin, permettent d'accéder à un flot comme à une séquence.

a) Itérateur en entrée

Un itérateur en entrée possède les mêmes propriétés qu'un itérateur unidirectionnel, avec cette différence qu'il n'autorise que la consultation de la valeur correspondante et plus sa modification ; si it est un tel itérateur :

```
... = *it ;     /* correct si it est un itérateur en entrée    */
*it = ... ;     /* impossible si it est un itérateur en entrée */
```

En outre, un itérateur en entrée n'autorise qu'un seul passage (on dit aussi une seule passe) sur les éléments qu'il permet de décrire. Autrement dit, si, à un moment donné, it1==it2, it1++ et it2++ ne désignent pas nécessairement la même valeur. Cette restriction n'existait pas dans le cas des itérateurs unidirectionnels. Ici, elle se justifie dès lors qu'on sait que l'itérateur en entrée est destiné à la lecture d'une suite de valeurs de même type sur un flot, d'une façon analogue à la lecture des informations d'une séquence. Or, manifestement, il n'est pas possible de lire deux fois une même valeur sur certains flots tels que l'unité d'entrée standard.

b) Itérateur en sortie

De façon concomitante, un itérateur en sortie possède les mêmes propriétés qu'un itérateur unidirectionnel, avec cette différence qu'il n'autorise que la modification et en aucun cas la consultation. Par exemple, si it est un tel itérateur :

```
*it = ... ;     /* correct si it est un itérateur en sortie    */
... = *it ;     /* impossible si it est un itérateur en sortie */
```

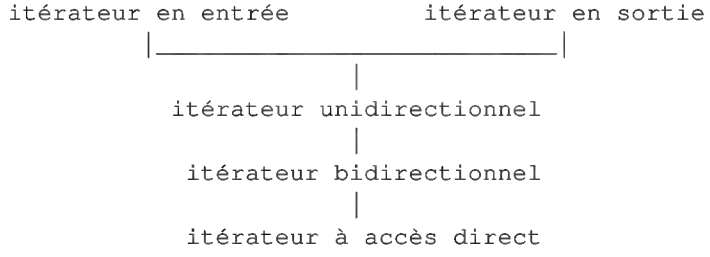
Comme l'itérateur en entrée, l'itérateur en sortie ne permettra qu'un seul passage ; si, à un moment donné, on a it1==it2, les affectations successives :

```
*it1++ = ... ;
*it2++ = ... ;
```

entraîneront la création de deux valeurs distinctes. Là encore, pour mieux comprendre ces restrictions, il faut voir que la principale justification de l'itérateur en sortie est de permettre d'écrire une suite de valeur de même type sur un flot, de la même façon qu'on peut introduire des informations dans une séquence. Or, manifestement, il n'est pas possible d'écrire deux fois en un même endroit de certains flots tels que l'unité standard de sortie.

c) Hiérarchie des catégories d'itérateurs

On peut montrer que les propriétés des cinq catégories d'itérateurs permettent de les ranger selon une hiérarchie dans laquelle toute catégorie possède au moins les propriétés de la catégorie précédente :



Les itérateurs en entrée et en sortie seront fréquemment utilisés pour associer un itérateur à un flot, en faisant appel à un adaptateur particulier d'itérateur dit itérateur de flot ; nous y reviendrons au paragraphe

1.5. En dehors de cela, ils présentent un intérêt indirect à propos de l'information qu'on peut déduire au vu de la catégorie d'itérateur attendu par un algorithme ; par exemple, si un algorithme accepte un itérateur en entrée, c'est que, d'une part, il ne modifie pas la séquence correspondante et que, d'autre part, il n'effectue qu'une seule passe sur cette séquence.

1.3 Algorithmes et séquences

Beaucoup d'algorithmes s'appliquent à une séquence définie par un intervalle d'itérateur de la forme [début, fin) ; dans ce cas, on lui communiquera simplement en argument les deux valeurs début et fin, lesquelles devront naturellement être du même type, sous peine d'erreur de compilation.

Tant que l'algorithme ne modifie pas les éléments de cette séquence, cette dernière peut appartenir à un conteneur de n'importe quel type, y compris les conteneurs associatifs pour lesquels, rappelons-le, la notion de séquence a bien un sens, compte tenu de leur ordre interne. Cependant, dans le cas des types map ou multimap, on sera généralement gêné par le fait que leurs éléments sont des paires.

En revanche, si l'algorithme modifie les éléments de la séquence, il n'est plus possible qu'elle appartienne à un conteneur de type set et multiset, puisque les éléments n'en sont plus modifiables. Bien qu'il n'existe pas d'interdiction formelle, il n'est guère raisonnable qu'elle appartienne à un conteneur de type map ou multimap, compte tenu des risques d'incompatibilité qui apparaissent alors entre l'organisation interne et celle qu'on chercherait à lui imposer...

Certains algorithmes s'appliquent à deux séquences de même taille. C'est par exemple le cas de la recopie d'une séquence dans une autre ayant des éléments de même type. Dans ce cas, tous ces algorithmes procèdent de la même façon, à savoir :

- deux arguments définissent classiquement un premier intervalle, correspondant à la première séquence,
- un troisième argument fournit la valeur d'un itérateur désignant le début de la seconde séquence.

On notera bien que cette façon de procéder présente manifestement le risque que la séquence cible soit trop petite. Dans ce cas, le comportement du programme est indéterminé comme il pouvait l'être en cas de débordement d'un tableau classique ; d'ailleurs, rien n'interdit de fournir à un algorithme un itérateur qui soit un pointeur...

Enfin, quelques rares algorithmes fournissent comme valeur de retour, les limites d'un intervalle, sous forme de deux itérateurs ; dans ce cas, celles-ci seront regroupées au sein d'une structure de type pair.

1.4 Itérateur d'insertion

Beaucoup d'algorithmes sont prévus pour modifier les valeurs des éléments d'une séquence ; c'est par exemple le cas de copy :

```
copy (v.begin(), v.end(), l.begin());/* recopie l'intervalle [v.begin(),
v.end() ) */
                                     /* à partir de la position l.begin()
*/
```

De telles opérations imposent naturellement un certain nombre de contraintes :

- les emplacements nécessaires à la copie doivent déjà exister,
- leur modification doit être autorisée, ce qui n'est pas le cas pour des conteneurs de type set ou multiset,
- la copie ne doit pas se faire à l'intérieur d'un conteneur associatif de type map ou multimap, compte tenu de l'incompatibilité qui résulterait entre l'ordre séquentiel imposé et l'ordre interne du conteneur.

En fait, il existe un mécanisme particulier permettant de transformer une succession d'opérations de copie à partir d'une position donnée en une succession d'insertions à partir de cette position. Pour ce faire, on fait

appel à ce qu'on nomme un itérateur d'insertion ; il s'agit d'un patron de classes nommé insert_iterator et paramétré par un type de conteneur. Par exemple :

```
insert_iterator <list<int> > ins ;   /* ins est un itérateur d'insertion dans
*/
                                     /* un conteneur de type  list<int>
*/
```

Pour affecter une valeur à un tel itérateur, on se sert du patron de fonction inserter ; en voici un exemple dans lequel on suppose que c est un conteneur et it est une valeur particulière d'itérateur sur ce conteneur :

```
ins = inserter(c, it) ;/* valeur initiale d'un itérateur d'insertion permettant
*/
                       /* d'insérer à partir de la position it dans le
conteneur c */
```

Dans ces conditions, l'utilisation de ins, en lieu et place d'une valeur initiale d'itérateur, fera qu'une instruction telle que *ins = ... insérera un nouvel élément en position it. De plus, toute incrémentation de ins, suivie d'une nouvelle affectation *ins=... provoquera une nouvelle insertion à la suite de l'élément précédent.

D'une manière générale, il existe trois fonctions permettant de définir une valeur initiale d'un itérateur d'insertion, à savoir :

- front_inserter (conteneur) : pour une insertion en début du conteneur ; le conteneur doit disposer de la fonction membre push_front ;
- back_inserter (conteneur) : pour une insertion en fin du conteneur ; le conteneur doit disposer de la fonction membre push_back ;
- inserter (conteneur, position) : pour une insertion à partir de position dans le conteneur ; le conteneur doit disposer de la fonction membre insert(valeur, position).

Voici un exemple de programme utilisant un tel mécanisme pour transformer une copie dans des éléments existant en une insertion ; auparavant, on a tenté une copie usuelle dans un conteneur trop petit pour montrer qu'elle se déroulait mal ; en pratique, nous déconseillons ce genre de procédé qui peut très bien amener à un plantage du programme :

```
#include <iostream.h>
#include <list>
using namespace std ;
main()
{ void affiche (list<char>) ;
  char t[] = {"essai insert_iterator"} ;
  list<char> l1(t, t+sizeof(t)-1) ;
  list<char> l2 (4, 'x') ;
  list<char> l3 ;
  cout << "l1 initiale            : " ; affiche(l1) ;
  cout << "l2 initiale            : " ; affiche(l2) ;
     /* copie avec liste l2 de taille insuffisante - deconseille en pratique */
  copy (l1.begin(), l1.end(), l2.begin()) ;
  cout << "l2 apres copie usuelle : " ; affiche(l2) ;
     /* insertion dans liste non vide ; pourrait utiliser aussi
front_inserter(l2) */
  copy (l1.begin(), l1.end(), inserter(l2, l2.begin())) ;
```

```
  cout << "l2 apres copie inser   : " ; affiche(l2) ;
     /* insertion dans liste vide ; on pourrait utiliser aussi */
     /* front_inserter(l3) ou back_inserter(l3)                */
  copy (l1.begin(), l1.end(), inserter(l3, l3.begin())) ;
  cout << "l3 apres copie inser   : " ; affiche(l3) ;
}
void affiche (list<char> l)
{ void af_car (char) ;
  for_each (l.begin(), l.end(), af_car) ;   /* appelle af_car pour chaque
element */
  cout << "\n" ;
}
void af_car (char c)
{ cout << c << " " ;
}                        ____________________

l1 initiale            : e s s a i   i n s e r t _ i t e r a t o r
l2 initiale            : x x x x
l2 apres copie usuelle : r r a t
l2 apres copie inser   : e s s a i   i n s e r t _ i t e r a t o r r r a t
l3 apres copie inser   : e s s a i   i n s e r t _ i t e r a t o r
```

Exemple d'utilisation d'un itérateur d'insertion

Remarque

Si l'on tient à mettre en évidence l'existence d'une classe insert_iterator, la simple instruction du précédent programme :

```
copy (l1.begin(), l1.end(), inserter(l2, l2.begin())) ;
```

peut se décomposer ainsi :

```
insert_iterator<list<char> > ins = inserter(l2, l2.begin()) ;
copy (l1.begin(), l1.end(), ins ) ;
```

1.5 Itérateur de flot

a) Itérateur de flot de sortie

Lorsqu'on lit, par exemple, sur l'entrée standard, une suite d'informations de même type, on peut considérer qu'on parcourt une séquence. Effectivement, il est possible de définir un itérateur sur une telle séquence ne disposant que des propriétés d'un itérateur d'entrée telles qu'elles ont été définies précédemment. Pour ce faire, il existe un patron de classes, nommé ostream_iterator, paramétré par le type des éléments concernés ; par exemple :

```
ostream_iterator<char>     /* type itérateur sur un flot d'entrée de caractères
*/
```

Cette classe dispose d'un constructeur recevant en argument un flot existant. C'est ainsi que :

```
ostream_iterator<char> flcar(cout) ;    /* flcar est un itérateur sur un flot
de */
```

```
                                        /* caractères connecté à cout
*/
```

Dans ces conditions, une instruction telle que :

```
*flcar = 'x' ;
```

envoie le caractère x sur le flot cout.

On notera qu'il est théoriquement possible d'incrémenter l'itérateur flcar en écrivant flcar++ ; cependant, une telle opération est sans effet car sans signification à ce niveau. Son existence est cependant précieuse puisqu'elle permettra d'utiliser un tel itérateur avec certains algorithmes standard, tels que copy.

Voici un exemple résumant ce que nous venons de dire :

```
#include <iostream.h>
#include <list>
using namespace std ;
main()
{ char t[] = {"essai iterateur de flot"} ;
  list<char> l(t, t+sizeof(t)-1) ;
  ostream_iterator<char> flcar(cout) ;
  *flcar = 'x' ; *flcar = '-' ;
  flcar++ ; flcar++ ;     /* pour montrer que l'incrementation est inoperante
ici */
  *flcar = ':' ;
  copy (l.begin(), l.end(), flcar) ;
}
```

```
x-:essai iterateur de flot
```

Exemple d'utilisation d'un itérateur de flot de sortie

Remarque

Ici, notre exemple s'appliquait à la sortie standard ; dans ces conditions, l'utilisation d'informations de type autre que char poserait le problème de leur séparation à l'affichage ou dans le fichier texte correspondant. En revanche, l'application à un fichier binaire quelconque ne poserait plus aucun problème.

b) Itérateur de flot d'entrée

De même qu'on peut définir des itérateurs de flot de sortie, on peut définir des itérateurs de flot d'entrée, suivant un procédé très voisin. Par exemple, avec :

```
istream_iterator<int> flint(cin) ;
```

on définit un itérateur nommé flint, sur un flot d'entrée d'entiers, connecté à cin. De la même manière, avec :

```
ifstream fich("essai", ios::in) ;
istream_iterator<int> flint(fich) ;
```

on définit un itérateur, nommé flint, sur un flot d'entrée d'entiers, connecté au flot fich, supposé convenablement ouvert.

Les itérateurs de flot d'entrée nécessitent cependant la possibilité d'en détecter la fin. Pour ce faire, il existe une convention permettant de construire un itérateur en représentant la fin, à savoir, l'utilisation d'un constructeur sans argument ; par exemple, avec :

```
istream_iterator<int> fin ;  /* fin est un itérateur représentant une fin de */
                             /* fichier sur un itérateur de flot d'entiers   */
```

Voici, par exemple, comment utiliser un itérateur de flot d'entrée pour recopier les informations d'un fichier dans une liste ; ici, nous créons une liste vide et nous utilisons un itérateur d'insertion pour y introduire le résultat de la copie :

```
list<int> l ;
ifstream fich("essai", ios::in) ;
istream_iterator<int, ptrdiff_t> flint(fich), fin ;
copy (flint, fin, inserter(l, l.begin())) ;
```

2. ALGORITHMES D'INITIALISATION DE SEQUENCES EXISTANTES

Tous ces algorithmes permettent de donner des valeurs à des éléments existant d'une séquence, dont la valeur est donc remplacée. De par leur nature même, ils ne sont pas adaptés aux conteneurs associatifs, à moins d'utiliser un itérateur d'insertion et de tenir compte de la nature de type pair de leurs éléments.

2.1 Copie d'une séquence dans une autre

Comme on l'a déjà vu à plusieurs reprises, on peut recopier une séquence dans une autre, pour peu que les types des éléments soient les mêmes. Par exemple, si l est une liste d'entiers et v un vecteur d'entiers :

```
copy (l.begin(), l.end(), v.begin()) ;  /* recopie les éléments de la liste l
dans */
                                        /* le vecteur v, à partir de son début
*/
```

Le sens de la copie est imposé, à savoir qu'on commence bien par recopier l.begin() en v.begin(). La seule contrainte (logique) qui soit imposée aux valeurs des itérateurs est que la position de la première copie n'appartienne pas à l'intervalle à copier. En revanche, rien n'interdirait, par exemple :

```
copy (v.begin()+1, v.begin()+10, v.begin());  /* recopie v[1] dans v[0]
*/
                                              /* v[2] dans v[1]... v[9] dans
v[8] */
```

Il existe également un algorithme copy_backward qui procède à la copie dans l'ordre inverse de copy, c'est-à-dire en commençant par le dernier élément. Dans ce cas, comme on peut s'y attendre, les itérateurs correspondants doivent être bidirectionnels.

Voici un exemple de programme utilisant copy pour réaliser des copies usuelles, ainsi que des insertions, par le biais d'un itérateur d'insertion :

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std ;
main()
{ int t[5] = { 1, 2, 3, 4, 5 } ;
  vector<int> v(t, t+5) ;    /* v contient : 1, 2, 3, 4, 5   */
  list<int> l(8, 0) ;        /* liste de 8 elements egaux a 0*/
  void affiche(vector<int>) ;
  void affiche(list<int>) ;
  cout << "liste initiale      : " ; affiche(l) ;
  copy (v.begin(), v.end(), l.begin()) ;
  cout << "liste apres copie 1 : " ; affiche(l) ;
  l.assign (3, 0) ;                 /* l contient maintenant 3 elements égaux
à 0 */
  copy (v.begin(), v.end(), l.begin()) ;     /* sequence trop courte :
deconseille */
  cout << "liste apres copie 2 : " ; affiche(l) ;
  l.erase(l.begin(), l.end()) ;                       /* l est maintenant vide
*/
  copy (v.begin(), v.end(), inserter(l, l.begin())) ; /* on y insere les elem
de v */
  cout << "liste apres copie 3 : " ; affiche(l) ;
}
void affiche(list<int> l)
{ list<int>::iterator il ;
  for (il=l.begin() ; il!=l.end() ; il++) cout << *il << " " ;
  cout << "\n" ;
}
```

```
liste initiale      : 0 0 0 0 0 0 0 0
liste apres copie 1 : 1 2 3 4 5 0 0 0
liste apres copie 2 : 5 2 3
liste apres copie 3 : 1 2 3 4 5
```

Exemple de copies usuelles et de copies avec insertion

2.2 Génération de valeurs par une fonction

Il est fréquent qu'on ait besoin d'initialiser un conteneur par des valeurs résultant d'un calcul. La bibliothèque standard offre un outil assez général à cet effet, à savoir ce qu'on nomme souvent un algorithme générateur. On lui fournit en argument, un objet fonction (il peut donc s'agir d'une fonction ordinaire) qu'il appellera pour déterminer la valeur à attribuer à chaque élément d'un intervalle. Une telle fonction ne reçoit aucun argument. Par exemple, l'appel :

```
generate (v.begin(), v.end(), suite) ;
```

utilisera la fonction suite pour donner une valeur à chacun des éléments de la séquence définie par l'intervalle [v.begin(), v.end()).

Voici un premier exemple faisant appel à une fonction ordinaire :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ int n = 10 ;
  vector<int> v(n, 0) ;  /* vecteur de n elements initialises a 0 */
  int suite() ;          /* fonction utilisee pour la generation d'entiers */
  void affiche(vector<int>) ;
  cout << "vecteur initial : " ; affiche(v) ;
  generate (v.begin(), v.end(), suite) ;
  cout << "vecteur genere  : " ; affiche(v) ;
}
int suite()
{ static int n = 0 ;
  return n++ ;
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}

vecteur initial : 0 0 0 0 0 0 0 0 0 0
vecteur genere  : 0 1 2 3 4 5 6 7 8 9
```

Génération de valeurs par une fonction ordinaire

On constate qu'il est difficile d'imposer une valeur initiale à la suite de nombres, autrement qu'en la fixant dans la fonction elle-même ; en particulier, il n'est pas possible de la choisir en argument. C'est là précisément que la notion de classe fonction s'avère intéressante comme le montre l'exemple suivant :

```
#include <iostream.h>
#include <vector>
using namespace std ;
class sequence         /* classe fonction utilisee pour la generation d'entiers
*/
{ public :
   sequence (int i) { n = i ;}            /* constructeur      */
   int operator() () { return n++ ; }     /* ne pas oublier () */
  private :
   int n ;                          /* valeur courante generee */
} ;
```

```
main()
{ int n = 10 ;
  vector<int> v(n, 0) ;  /* vecteur de n elements initialises a 0 */
  void affiche(vector<int>) ;
  cout << "vecteur initial  : " ; affiche(v) ;
  generate (v.begin(), v.end(), sequence(0)) ;
  cout << "vecteur genere 1 : " ; affiche(v) ;
  generate (v.begin(), v.end(), sequence(4)) ;
  cout << "vecteur genere 2 : " ; affiche(v) ;
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
vecteur initial  : 0 0 0 0 0 0 0 0 0 0
vecteur genere 1 : 0 1 2 3 4 5 6 7 8 9
vecteur genere 2 : 4 5 6 7 8 9 10 11 12 13
```

Génération de valeurs par une classe fonction

Remarques

1) Si l'on compare les deux appels suivants, l'un du premier exemple, l'autre du second :

```
generate (v.begin(), v.end(), suite) ;
generate (v.begin(), v.end(), sequence(0)) ;
```

on constate que, dans le premier cas, suite est la référence à une fonction, tandis que dans le second, sequence(0) est la référence à un objet de type sequence. Mais, comme ce dernier a convenablement surdéfini l'opérateur (), l'algorithme generate n'a pas à tenir compte de cette différence.

2) Il existe un autre algorithme, generate_n, comparable à generate, qui génère un nombre de valeurs prévues en argument. D'autre part, l'algorithme fill permet d'affecter une valeur donnée à tous les éléments d'une séquence ou à un nombre donné d'éléments :

fill (début, fin, valeur)

fill (position, NbFois, valeur)

3. ALGORITHMES DE RECHERCHE

Ces algorithmes ne modifient pas la séquence sur laquelle ils travaillent. On distingue :

- les algorithmes fondés sur une égalité ou sur un prédicat unaire,
- les algorithmes fondés sur une relation d'ordre permettant de trouver le plus grand ou le plus petit élément.

3.1 Algorithmes fondés sur une égalité ou un prédicat unaire

Ces algorithmes permettent de rechercher la première occurrence de valeurs ou de séries de valeurs qui sont :

- soit imposées explicitement ; cela signifie en fait qu'on se fonde sur la relation d'égalité induite par l'opérateur ==, qu'il soit surdéfini ou non ;
- soit par une condition fournie sous forme d'un prédicat unaire.

Ils fournissent tous un itérateur sur l'élément recherché, s'il existe, et l'itérateur sur la fin de la séquence, sinon ; dans ce dernier cas, cette valeur n'est égale à end() que si la séquence concernée appartient à un conteneur et s'étend jusqu'à sa fin. Sinon, on peut obtenir un itérateur valide sur un élément n'ayant rien à voir avec la recherche en question. Dans le cas où les itérateurs utilisés sont des pointeurs, on peut obtenir un pointeur sur une valeur située au-delà de la séquence examinée. Il faudra tenir compte de ces remarques dans le test de la valeur de retour, qui constitue le seul moyen de savoir si la recherche a abouti.

L'algorithme find permet de rechercher une valeur donnée, tandis que find_first_of permet de rechercher une valeur parmi plusieurs. L'algorithme find_if (début, fin, prédicat) autorise la recherche de la première valeur satisfaisant au prédicat unaire fourni en argument.

On peut rechercher, dans une séquence [début_1, fin_1), la première apparition complète d'une autre séquence [début_2, fin_2) par search (début_1, fin_1, début_2, fin_2). De même, search_n (début, fin, NbFois, valeur) permet de rechercher une suite de NbFois une même valeur. Là encore, on se base sur l'opérateur ==, surdéfini ou non.

On peut rechercher les "doublons", c'est-à-dire les valeurs apparaissant deux fois de suite, par adjacent_find (début, fin). Attention, ce n'est pas un cas particulier de search_n, dans la mesure où l'on n'impose pas la valeur dupliquée. Pour chercher les autres doublons, on peut soit supprimer l'une des valeurs trouvées, soit simplement recommencer la recherche, au-delà de l'emplacement où se trouve le doublon précédent.

Voici un exemple de programme illustrant la plupart de ces possibilités (par souci de simplification, nous supposons que les valeurs recherchées existent toujours) :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ char *ch1 = "anticonstitutionnellement" ;
  char *ch2 = "uoie" ;
  char *ch3 = "tion" ;
  vector<char> v1 (ch1, ch1+strlen(ch1)) ;
  vector<char> v2 (ch2, ch2+strlen(ch2)) ;
  vector<char>::iterator iv ;
  iv = find_first_of (v1.begin(), v1.end(), v2.begin(), v2.end()) ;
  cout << "\npremier de uoie en : " ; for ( ; iv!=v1.end() ; iv++) cout << *iv
;
  iv = find_first_of (v1.begin(), v1.end(), v2.begin(), v2.begin()+2) ;
  cout << "\npremier de uo en   : " ; for ( ; iv!=v1.end() ; iv++) cout << *iv
;
  v2.assign (ch3, ch3+strlen(ch3)) ;
  iv = search (v1.begin(), v1.end(), v2.begin(), v2.end()) ;
  cout << "\ntion en            : " ;   for ( ; iv!=v1.end() ; iv++) cout <<
*iv ;
  iv = search_n(v1.begin(), v1.end(), 2, 'l' ) ;
```

```
  cout << "\n'l' 2 fois en      : " ;   for ( ; iv!=v1.end() ; iv++) cout <<
*iv ;
  iv = adjacent_find(v1.begin(), v1.end()) ;
  cout << "\npremier doublon en : " ;   for ( ; iv!=v1.end() ; iv++) cout <<
*iv ;
}
```

```
premier de uoie en : iconstitutionnellement
premier de uo en   : onstitutionnellement
tion en            : tionnellement
'l' 2 fois en      : llement
premier doublon en : nnellement
```

Exemple d'utilisation des algorithmes de recherche

3.2 Algorithmes de recherche de maximum ou de minimum

Les deux algorithmes max_element et min_element permettent de déterminer le plus grand ou le plus petit élément d'une séquence. Ils s'appuient par défaut sur la relation induite par l'opérateur <, mais il est également possible d'imposer sa propre relation, sous forme d'un prédicat binaire. Comme les algorithmes précédents, ils fournissent en retour soit un itérateur sur l'élément correspondant ou sur le premier d'entre eux s'il en existe plusieurs, soit un itérateur sur la fin de la séquence, s'il n'en existe aucun. Mais cette dernière situation ne peut se produire ici qu'avec une séquence vide ou lorsqu'on choisit son propre prédicat, de sorte que l'examen de la valeur de retour est alors moins cruciale.

Voici un exemple dans lequel nous appliquons ces algorithmes à un tableau usuel (par souci de simplification, nous supposons que les valeurs recherchées existent toujours) :

```
#include <iostream.h>
#include <algorithm>   // utile car aucun en-tete de conteneur
using namespace std ;
main()
{ int t[] = {5, 4, 1, 8, 3, 9, 2, 9, 1, 8} ;
  int * ad ;
  ad = max_element(t, t+sizeof(t)/sizeof(t[0])) ;
  cout << "plus grand elem de t en position " << ad-t
       << " valeur " << *ad << "\n" ;
  ad = min_element(t, t+sizeof(t)/sizeof(t[0])) ;
  cout << "plus petit elem de t en position " << ad-t
       << " valeur " << *ad << "\n" ;
  ad = max_element(t, t+sizeof(t)/sizeof(t[0]), greater<int>()) ;
  cout << "plus grand elem avec greater<int> en position " << ad-t
       << " valeur " << *ad << "\n" ;
}
```

```
plus grand elem de t en position 5 valeur 9
plus petit elem de t en position 2 valeur 1
plus grand elem avec greater<int> en position 2 valeur 1
```

Exemple d'utilisation de max_element et de min_element

4. ALGORITHMES DE TRANSFORMATION D'UNE SEQUENCE

Il s'agit des algorithmes qui modifient les valeurs d'une séquence ou leur ordre, sans en modifier le nombre d'éléments. Ils ne sont pas applicables aux conteneurs associatifs, pour lesquels l'ordre est imposé de façon intrinsèque.

On peut distinguer trois catégories d'algorithmes :

- remplacement de valeurs,
- permutation de valeurs,
- partition.

Beaucoup de ces algorithmes disposent d'une version suffixée par _copy ; dans ce cas, la version xxxx_copy réalise le même traitement que xxxx, avec cette différence importante qu'elle ne modifie plus la séquence d'origine et qu'elle copie le résultat obtenu dans une autre séquence dont les éléments doivent alors exister, comme avec copy. Ces algorithmes de la forme xxxx_copy peuvent, quant à eux, s'appliquer à des conteneurs associatifs, à condition toutefois, d'utiliser un itérateur d'insertion et de tenir compte de la nature de type pair de leurs éléments.

Par ailleurs, il existe un algorithme nommé transform qui, contrairement à ce que son nom pourrait laisser entendre, initialise une séquence en appliquant une fonction de transformation à une séquence ou à deux séquences de même taille, ces dernières n'étant alors pas modifiées.

4.1 Remplacement de valeurs

On peut remplacer toutes les occurrences d'une valeur donnée par une autre valeur, en se fondant sur l'opérateur == ; par exemple :

```
replace (l.begin(), l.end(), 0, -1) ;    /* remplace toutes les occurrences */
                                         /*  de 0 par -1                   */
```

On peut également remplacer toutes les occurrences d'une valeur satisfaisant à une condition ; par exemple :

```
replace_if (l.begin(), l.end(), impair, 0) ;  /* remplace par 0 toutes les
valeurs */
                                              /* satisfaisant au prédicat
unaire   */
                                              /* impair qu'il faut fournir
*/
```

4.2 Permutations de valeurs

a) Rotation

L'algorithme rotate permet d'effectuer une permutation circulaire des valeurs d'une séquence. On notera qu'on ne dispose que des possibilités de permutation circulaire inverse compte tenu de la manière dont on

précise l'ampleur de la permutation, à savoir, non pas par un nombre, mais en indiquant quel élément doit venir en première position. En voici un exemple :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ void affiche (vector<int>) ;
  int t[] = {1, 2, 3, 4, 5, 6, 7, 8} ;
  int decal = 3 ;
  vector<int> v(t, t+8) ;
  cout << "vecteur initial     : " ; affiche(v) ;
  rotate (v.begin(), v.begin()+decal, v.end()) ;
  cout << "vecteur decale de 3 : " ; affiche(v) ;
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
vecteur initial     : 1 2 3 4 5 6 7 8
vecteur decale de 3 : 4 5 6 7 8 1 2 3
```

Exemple d'utilisation de rotate

b) Génération de permutations

Dès lors qu'une séquence est ordonnée par une relation d'ordre R, il est possible d'ordonner les différentes permutations possibles des valeurs de cette séquence. Par exemple, si l'on considère les trois valeurs 1, 4, 8 et la relation d'ordre <, voici la liste ordonnée de toutes les permutations possibles :

```
148
184
418
481
814
841
```

Dans ces conditions, il est possible de parler de la permutation suivante ou précédente d'une séquence de valeurs données. Dans l'exemple ci-dessus, la permutation précédente de la séquence 4, 1, 8 serait la séquence 1, 8, 4 tandis que la permutation suivante serait 4, 8, 1. Pour éviter tout problème, on considère que la permutation suivant la dernière est la première, et que la permutation précédent la dernière est la première.

Les algorithmes next_permutation et prev_permutation permettent de remplacer une séquence donnée respectivement par la permutation suivante ou par la permutation précédente. On peut utiliser soit, par défaut, l'opérateur <, soit une relation imposée sous forme d'un prédicat binaire. Actuellement, il n'existe pas de variantes _copy de ces algorithmes.

Voici un exemple (la valeur de retour true ou false des algorithmes permet de savoir si l'on a effectué un bouclage dans la liste des permutations) :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ void affiche (vector<int>) ;
  int t[] = {2, 1, 3} ;
  int i ;
  vector<int> v(t, t+3) ;
  cout << "vecteur initial   : " ; affiche(v) ;
  for (i=0 ; i<=10 ; i++)
  { bool res = next_permutation (v.begin(), v.end()) ;
    cout << "permutation " << res << "     : " ; affiche(v) ;
  }
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
vecteur initial   : 2 1 3
permutation 1     : 2 3 1
permutation 1     : 3 1 2
permutation 1     : 3 2 1
permutation 0     : 1 2 3
permutation 1     : 1 3 2
permutation 1     : 2 1 3
permutation 1     : 2 3 1
permutation 1     : 3 1 2
permutation 1     : 3 2 1
permutation 0     : 1 2 3
permutation 1     : 1 3 2
```

Exemple d'utilisation de next_permutation et de prev_permutation

c) Permutations aléatoires

L'algorithme random_shuffle permet d'effectuer une permutation aléatoire des valeurs d'une séquence. En voici un exemple :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ void affiche (vector<int>) ;
  int t[] = {2, 1, 3} ;
  int i ;
```

```
    vector<int> v(t, t+3) ;
    cout << "vecteur initial : " ; affiche(v) ;
    for (i=0 ; i<=10 ; i++)
    { random_shuffle (v.begin(), v.end()) ;
      cout << "vecteur hasard  : " ; affiche(v) ;
    }
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
vecteur initial : 2 1 3
vecteur hasard  : 3 2 1
vecteur hasard  : 2 3 1
vecteur hasard  : 1 2 3
vecteur hasard  : 1 3 2
vecteur hasard  : 3 1 2
vecteur hasard  : 3 2 1
vecteur hasard  : 3 1 2
vecteur hasard  : 3 2 1
vecteur hasard  : 2 3 1
vecteur hasard  : 2 3 1
vecteur hasard  : 2 3 1
```

Exemple d'utilisation de random_shuffle

Remarque

Il existe une version de random_shuffle permettant d'imposer son générateur de nombres aléatoires.

4.3 Partitions

On nomme partition d'une séquence suivant un prédicat unaire donné, un réarrangement de cette séquence défini par un itérateur désignant un élément tel que tous les éléments le précédant vérifient la dite condition. Par exemple, avec la séquence :

1 3 4 11 2 7 8

et le prédicat impair (supposé vrai pour un nombre impair et faux sinon), voici des partitions possibles (dans tous les cas, l'itérateur désignera le quatrième élément) :

```
1 3 11 7 4 2 8     /* l'itérateur désignera ici le 4 */
1 3 11 7 2 8 4     /* l'itérateur désignera ici le 2 */
3 1 7 11 2 4 8     /* l'itérateur désignera ici le 2 */
```

On dit que la partition obtenue est stable si l'ordre relatif des éléments satisfaisant au prédicat est conservé. Dans notre exemple, seules les deux premières permutations sont stables.

Les algorithmes partition et stable_partition permettent de déterminer une telle partition à partir d'un prédicat unaire fourni en argument.

5. ALGORITHMES DITS "DE SUPPRESSION"

Ces algorithmes permettent d'éliminer d'une séquence les éléments répondant à un certain critère. Mais, assez curieusement, ils ne suppriment pas les éléments correspondants ; ils se contentent de regrouper en début de séquence les éléments non concernés par la condition d'élimination et de fournir en retour un itérateur sur le premier élément non conservé. En fait, il faut voir qu'aucun algorithme ne peut supprimer des éléments d'une séquence pour la bonne et simple raison qu'il risque d'être appliqué à une structure autre qu'un conteneur (ne serait-ce qu'un tableau usuel) pour laquelle la notion de suppression n'existe pas[1]. D'autre part, contrairement à toute attente, il n'est pas du tout certain que les valeurs apparaissant en fin de conteneur soient celles qui ont été éliminées du début.

Bien entendu, rien n'empêche d'effectuer, après avoir appelé un tel algorithme, une suppression effective des éléments concernés en utilisant une fonction membre telle que remove, dans le cas où l'on a affaire à une séquence d'un conteneur.

L'algorithme remove (début, fin, valeur) permet d'éliminer tous les éléments ayant la valeur indiquée, en se basant sur l'opérateur ==. Il existe une version remove_if, qui se fonde sur un prédicat binaire donné. Seul le premier algorithme est stable, c'est-à-dire qu'il conserve l'ordre relatif des valeurs non éliminées.

L'algorithme unique permet de ne conserver que la première valeur d'une série de valeurs égales (au sens de ==) ou répondant à un prédicat binaire donné. Il n'impose nullement que la séquence soit ordonnée suivant un certain ordre.

Ces algorithmes disposent d'une version avec _copy qui ne modifie pas la séquence d'origine et qui range dans une autre séquence les seules valeurs non éliminées. Utilisés conjointement avec un itérateur d'insertion, ils peuvent permettre de créer une nouvelle séquence.

Voici un exemple de programme montrant les principales possibilités évoquées, y compris des insertions dans une séquence avec remove_copy_if (dont on remarque clairement d'ailleurs qu'il n'est pas stable) :

```
#include <iostream.h>
#include <list>
using namespace std ;
main()
{ void affiche(list<int>) ;
  bool pair (int) ;
  int t[] = { 4, 3, 5, 4, 4, 4, 9, 4, 6, 6, 3, 3, 2 } ;
  list<int> l (t, t+sizeof(t)/sizeof(int)) ;
  list<int> l_bis=l ;
  list<int> l2 ;     /* liste vide */
  list<int>::iterator il ;
  cout << "liste initiale                : " ; affiche(l) ;

  il = remove(l.begin(), l.end(), 4) ;  /* different de l.remove(4) */
  cout << "liste apres remove(4)         : " ; affiche(l) ;
  cout << "element places en fin         : " ;
  for (; il!=l.end() ; il++) cout << *il << " " ; cout << "\n" ;
```

1 - Un algorithme ne peut pas davantage insérer un élément dans une séquence ; on peut toutefois y parvenir, dans le cas d'un conteneur, en recourant à un itérateur d'insertion.

```
    l = l_bis ;
    il = unique (l.begin(), l.end()) ;
    cout << "liste apres unique             : " ; affiche(l) ;
    cout << "elements places en fin         : " ;
    for (; il!=l.end() ; il++) cout << *il << " " ; cout << "\n" ;

    l = l_bis ;
    il = remove_if(l.begin(), l.end(), pair) ;
    cout << "liste apres remove pair        : " ; affiche(l) ;
    cout << "elements places en fin         : " ;
    for (; il!=l.end() ; il++) cout << *il << " " ; cout << "\n" ;

    /* elimination de valeurs par copie dans liste vide l2 par iterateur
  d'insertion */
    l = l_bis ;
    remove_copy_if(l.begin(), l.end(), front_inserter(l2), pair) ;
    cout << "liste avec remove_copy_if pair : " ; affiche(l2) ;
  }
  void affiche(list<int> l)
  { list<int>::iterator il ;
    for (il=l.begin() ; il!=l.end() ; il++) cout << (*il) << " " ;
    cout << "\n" ;
  }
  bool pair (int n)
  { return !(n%2) ;
  }
```

```
liste initiale                 : 4 3 5 4 4 4 9 4 6 6 3 3 2
liste apres remove(4)          : 3 5 9 6 6 3 3 2 6 6 3 3 2
element places en fin          : 6 6 3 3 2
liste apres unique             : 4 3 5 4 9 4 6 3 2 6 3 3 2
elements places en fin         : 6 3 3 2
liste apres remove pair        : 3 5 9 3 3 4 9 4 6 6 3 3 2
elements places en fin         : 4 9 4 6 6 3 3 2
liste avec remove_copy_if pair : 3 3 9 5 3
```

Exemple d'utilisation des algorithmes de suppression

6. ALGORITHMES DE TRIS

Ces algorithmes s'appliquent à des séquences ordonnables, c'est-à-dire pour lesquelles il a été défini une relation d'ordre faible strict, soit par l'opérateur <, soit par un prédicat binaire donné. Ils ne peuvent pas s'appliquer à un conteneur associatif, compte tenu du conflit qui apparaîtrait alors entre leur ordre interne et celui qu'on voudrait leur imposer. Pour d'évidentes questions d'efficacité, la plupart de ces algorithmes nécessitent des itérateurs à accès direct, de sorte qu'ils ne sont pas applicables à des listes (mais le conteneur list dispose de sa fonction membre sort).

On peut réaliser des tris complets d'une séquence. Dans ce cas, on peut choisir entre un algorithme stable stable_sort ou un algorithme non stable, plus rapide. On peut effectuer également, avec partial_sort, des tris partiels, c'est-à-dire qui se contentent de n'ordonner qu'un certain nombre d'éléments. Dans ce cas, l'appel se présente sous la forme partial_sort (début, milieu, fin) et l'amplitude du tri est définie par l'itérateur milieu désignant le premier élément non trié. Enfin, avec nth_element, il est possible de déterminer seulement le nième élément, c'est-à-dire de placer dans cette position l'élément qui s'y trouverait si l'on avait trié toute la séquence ; là encore, l'appel se présente sous la forme nth_element (début, milieu, fin) et milieu désigne l'élément en question.

Voici un exemple montrant l'utilisation des principaux algorithmes de tri :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ void affiche (vector<int>) ;
  bool comp (int, int) ;
  int t[] = {2, 1, 3, 9, 2, 7, 5, 8} ;
  int i ;
  vector<int> v(t, t+8), v_bis=v ;
  cout << "vecteur initial        : " ; affiche(v) ;
  sort (v.begin(), v.end()) ;
  cout << "apres sort             : " ; affiche(v) ;
  v = v_bis ;
  partial_sort (v.begin(), v.begin()+5, v.end()) ;
  cout << "apres partial_sort (5) : " ; affiche(v) ;
  v = v_bis ;
  nth_element (v.begin(), v.begin()+ 5, v.end()) ;
  cout << "apres nth_element 6    : " ; affiche(v) ;
  nth_element (v.begin(), v.begin()+ 2, v.end()) ;
  cout << "apres nth_element 3    : " ; affiche(v) ;
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
vecteur initial        : 2 1 3 9 2 7 5 8
apres sort             : 1 2 2 3 5 7 8 9
apres partial_sort (5) : 1 2 2 3 5 9 7 8
apres nth_element 6    : 2 1 3 5 2 7 8 9
apres nth_element 3    : 2 1 2 3 5 7 8 9
```

Exemple d'utilisation des algorithmes de tri

7. ALGORITHMES DE RECHERCHE ET DE FUSION SUR DES SEQUENCES ORDONNEES

Ces algorithmes s'appliquent à des séquences supposées ordonnées par une relation d'ordre faible strict.

7.1 Algorithmes de recherche binaire

Les algorithmes de recherche présentés dans le paragraphe 3 s'appliquaient à des séquences non nécessairement ordonnées. Les algorithmes présentés ici supposent que la séquence concernée soit convenablement ordonnée suivant la relation d'ordre faible strict qui sera utilisée, qu'il s'agisse par défaut de l'opérateur < ou d'un prédicat fourni explicitement. C'est ce qui leur permet d'utiliser des méthodes de recherche dichotomique (ou binaire) plus performantes que de simples recherches séquentielles.

Comme on peut s'y attendre, ces algorithmes ne modifient pas la séquence concernée et ils peuvent donc, en théorie, s'appliquer à des conteneurs de type set ou multiset. En revanche, leur application à des types map et multimap n'est guère envisageable puisque, en général, ce ne sont pas leurs éléments qui sont ordonnés, mais seulement les clés... Quoi qu'il en soit, les conteneurs associatifs disposent déjà de fonctions membre équivalant aux algorithmes examinés ici, excepté pour binary_search.

L'algorithme binary_search permet de savoir s'il existe dans la séquence une valeur équivalente (au sens de l'équivalence induite par la relation d'ordre concernée). Par ailleurs, on peut localiser l'emplacement possible pour une valeur donnée, compte tenu d'un certain ordre : lower_bound fournit la première position possible tandis que upper_bound fournit la dernière position possible ; equal_range fournit les deux informations précédentes sous forme d'une paire.

7.2 Algorithmes de fusion

La fusion de deux séquences ordonnées consiste à les réunir en une troisième séquence ordonnée suivant le même ordre. Là encore, ils peuvent s'appliquer à des conteneurs de type set ou multiset ; en revanche, leur application à des conteneurs de type map ou multimap n'est guère réaliste, compte tenu de ce que ces derniers sont ordonnés uniquement suivant les clés. Il existe deux algorithmes :

- merge qui permet la création d'une troisième séquence par fusion de deux autres ;
- inplace_merge qui permet la fusion de deux séquences consécutives en une seule qui vient prendre la place des deux séquences originales.

Voici un exemple d'utilisation de ces algorithmes :

```
#include <iostream.h>
#include <vector>
using namespace std ;
main()
{ void affiche (vector<int>) ;
  int t1[8] = {2, 1, 3, 12, 2, 18, 5, 8} ;
  int t2[5] = {5, 4, 15, 9, 11} ;
  vector<int> v1(t1, t1+8), v2(t2, t2+6), v ;
  cout << "vecteur 1 initial          : " ; affiche(v1) ;
  sort (v1.begin(), v1.end()) ;
  cout << "vecteur 1 trie             : " ; affiche(v1) ;
  cout << "vecteur 2 initial          : " ; affiche(v2) ;
  sort (v2.begin(), v2.end()) ;
  cout << "vecteur 2 trie             : " ; affiche(v2) ;
```

```
  merge (v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(v)) ;
  cout << "fusion des deux            : " ; affiche(v) ;
  random_shuffle (v.begin(), v.end()) ;  /* v n'est plus ordonne */
  cout << "vecteur v desordonne       : " ; affiche(v) ;
  sort (v.begin(), v.begin()+6) ;        /* tri des premiers elements de v */
  sort (v.begin()+6, v.end()) ;          /* tri des derniers elements de v */
  cout << "vecteur v trie par parties : " ; affiche(v) ;
  inplace_merge (v.begin(), v.begin()+6, v.end()) ; /* fusion interne */
  cout << "vecteur v apres fusion     : " ; affiche(v) ;
}
void affiche (vector<int> v)
{ unsigned int i ;
  for (i=0 ; i<v.size() ; i++)
    cout << v[i] << " " ;
  cout << "\n" ;
}
```

```
vecteur 1 initial          : 2 1 3 12 2 18 5 8
vecteur 1 trie             : 1 2 2 3 5 8 12 18
vecteur 2 initial          : 5 4 15 9 11 2
vecteur 2 trie             : 2 4 5 9 11 15
fusion des deux            : 1 2 2 2 3 4 5 5 8 9 11 12 15 18
vecteur v desordonne       : 5 12 9 2 2 15 2 5 1 18 3 8 11 4
vecteur v trie par parties : 2 2 5 9 12 15 1 2 3 4 5 8 11 18
vecteur v apres fusion     : 1 2 2 2 3 4 5 5 8 9 11 12 15 18
```

Exemple d'utilisation des algorithmes de fusion

8. ALGORITHMES A CARACTERE NUMERIQUE

Nous avons classé dans cette rubrique les algorithmes qui effectuent, sur les éléments d'une séquence, des opérations numériques fondées sur les opérateurs +, - ou *. Plutôt destinés, a priori, à des éléments d'un type effectivement numérique, ils peuvent néanmoins s'appliquer à des éléments de type classe pour peu que cette dernière ait convenablement surdéfini les opérateurs voulus ou qu'elle fournisse une fonction binaire appropriée.

Comme on peut s'y attendre, l'algorithme accumulate fait la somme des éléments d'une séquence tandis que inner_product effectue le produit scalaire de deux séquences de même taille. On prendra garde au fait que ces deux algorithmes ajoutent le résultat à une valeur initiale fournie en argument (en général, on choisit 0).

L'algorithme partial_sum crée, à partir d'une séquence, une nouvelle séquence de même taille formée des cumuls partiels des valeurs de la première : le premier élément est inchangé, le second est la somme du premier et du second, etc. Enfin, l'algorithme adjacent_difference crée, à partir d'une séquence, une séquence de même taille formée des différences de deux éléments consécutifs (le premier élément restant inchangé).

Voici un exemple d'utilisation de ces différents algorithmes :

```
#include <iostream.h>
#include <numeric>    // pour les algorithmes numeriques
using namespace std ;
```

```
main()
{ void affiche (int *) ;
  int v1[5] = { 1, 3, -1, 4,  1} ;
  int v2[5] = { 2, 5,  1, -3, 2} ;
  int v3[5] ;
  cout << "vecteur v1                    : " ; affiche(v1) ;
  cout << "vecteur v2                    : " ; affiche(v2) ;
  cout << "somme des elements de v1      : "
       << accumulate (v1, v1+3, 0) << "\n"  ;         /* ne pas oublier 0 */
  cout << "produit scalaire v1.v2        : "
       << inner_product (v1, v1+3, v2, 0) << "\n"  ;  /* ne pas oublier 0 */
  partial_sum (v1, v1+5, v3) ;
  cout << "sommes partielles de v     1  : " ; affiche(v3) ;
  adjacent_difference (v1, v1+5, v3) ;
  cout << "differences ajdacentes de v1  : " ; affiche(v3) ;
}
void affiche (int * v)
{ int i ; for (i=0 ; i<5 ; i++) cout << v[i] << " " ; cout << "\n" ;
}
```

```
vecteur v1                    : 1 3 -1 4 1
vecteur v2                    : 2 5 1 -3 2
somme des elements de v1      : 3
produit scalaire v1.v2        : 16
sommes partielles de v     1  : 1 4 3 7 8
differences ajdacentes de v1  : 1 2 -4 5 -3
```

Exemple d'utilisation d'algorithmes numériques

9. ALGORITHMES A CARACTERE ENSEMBLISTE

Comme on a pu le constater dans le chapitre précédent, les conteneurs set et multiset ne disposent d'aucune fonction membre permettant de réaliser les opérations ensemblistes classiques. En revanche, il existe des algorithmes généraux qui, quant à eux, peuvent en théorie s'appliquer à des séquences quelconques ; il faut cependant qu'elles soient convenablement ordonnées, ce qui constitue une première différence par rapport aux notions mathématiques usuelles, dont l'ordre est manifestement absent. De plus, ces notions ensemblistes ont dû être quelque peu aménagées, de manière à accepter la présence de plusieurs éléments de même valeur.

L'égalité entre deux éléments se fonde sur l'opérateur == ou, éventuellement, sur un prédicat binaire fourni explicitement. Pour que les algorithmes fonctionnent convenablement, il est alors nécessaire que cette relation d'égalité soit compatible avec la relation ayant servi à ordonner les séquences correspondantes ; plus précisément, il est nécessaire que les classes d'équivalence induite par la relation d'ordre faible strict coïncident avec celles qui sont induites par l'égalité.

Par ailleurs, les algorithmes créant une nouvelle séquence le font, comme toujours, dans des éléments existants, ce qui pose manifestement un problème avec des conteneurs de type set ou multiset qui n'autorisent pas la modification des valeurs de leurs éléments mais seulement les suppressions ou les insertions. Dans ce cas, il faudra donc recourir à un itérateur d'insertion pour la séquence à créer. De plus, comme ni set ni multiset ne disposent d'insertion en début ou en fin, cet itérateur d'insertion ne pourra être que inserter.

Voici un exemple correspondant à l'usage le plus courant des algorithmes, à savoir leur application à des conteneurs de type set.

```
#include <iostream.h>
#include <set>
using namespace std ;
main()
{ char t1[] = "je me figure ce zouave qui joue du xylophone" ;
  char t2[] = "en buvant du whisky" ;
  void affiche (set<char> ) ;
  set<char> e1(t1, t1+sizeof(t1)-1) ;
  set<char> e2(t2, t2+sizeof(t2)-1) ;
  set<char> u, i, d, ds ;
  cout << "ensemble 1     : " ; affiche (e1) ;
  cout << "ensemble 2     : " ; affiche (e2) ;
  set_union (e1.begin(), e1.end(), e2.begin(), e2.end(),
             inserter(u, u.begin())) ;
  cout << "union des deux : " ; affiche (u) ;
  set_intersection (e1.begin(), e1.end(), e2.begin(), e2.end(),
                    inserter(i, i.begin())) ;
  cout << "intersecton des deux           : " ; affiche (i) ;
  set_difference (e1.begin(), e1.end(), e2.begin(), e2.end(),
                  inserter(d, d.begin())) ;
  cout << "difference des deux            : " ; affiche (d) ;
  set_symmetric_difference (e1.begin(), e1.end(), e2.begin(), e2.end(),
                            inserter(ds, ds.begin())) ;
  cout << "difference_symetrique des deux : " ; affiche (ds) ;
}
void affiche (set<char> e )
{ set<char>::iterator ie ;
  for (ie=e.begin() ; ie!=e.end() ; ie++)  cout << *ie << " " ;  cout << "\n" ;
}
```

```
ensemble 1     :   a c d e f g h i j l m n o p q r u v x y z
ensemble 2     :   a b d e h i k n s t u v w y
union des deux :   a b c d e f g h i j k l m n o p q r s t u v w x y z
intersecton des deux           :   a d e h i n u v y
difference des deux            : c f g j l m o p q r x z
difference_symetrique des deux : b c f g j k l m o p q r s t w x z
```

Exemple d'utilisation d'algorithmes à caractère ensembliste avec un conteneur de type set

XXII. LA CLASSE STRING

Si l'on cherche à manipuler des chaînes de caractères en se fondant uniquement sur les instructions de base du langage C++, les choses ne sont pas plus satisfaisantes qu'en C ; en particulier on n'y dispose pas d'un type chaîne à part entière et même une opération aussi banale que l'affectation n'existe pas ; quant aux possibilités de gestion dynamique, on ne peut y accéder qu'en gérant soi même les choses...

La bibliothèque standard dispose d'un patron de classes permettant de manipuler des chaînes généralisées, c'est-à-dire des suites de valeurs de type quelconque donc, en particulier, de type char. Il s'agit du patron basic_string paramétré par le type des éléments. Mais il existe une classe spécialisée nommée string qui est définie comme basic_string< char> . Ici, nous nous limiterons à l'examen des propriétés de cette classe qui est de loin la plus utilisée ; la généralisation à basic_string ne présente, de toutes façons, aucune difficulté.

La classe string propose un cadre très souple de manipulation de chaînes de caractères en offrant les fonctionnalités traditionnelles qu'on peut attendre d'un tel type : gestion dynamique transparente des emplacements correspondants, affectation, concaténation, recherche de sous-chaînes, insertions ou suppression de sous-chaînes... On verra qu'elle possède non seulement beaucoup des fonctionnalités de la classe vector (plus précisément vector< char> pour string), mais également bien d'autres. D'une manière générale, ces fonctionnalités se mettent en œuvre de façon très naturelle, ce qui nous permettra de les présenter assez brièvement. Il faut cependant noter une petite difficulté liée à la présence de certaines possibilités redondantes, les unes faisant appel à des itérateurs usuels, les autres à des valeurs d'indices.

1. GENERALITES

Un objet de type string contient, à un instant donné, une suite formée d'un nombre quelconque de caractères quelconques. Sa taille peut évoluer dynamiquement au fil de l'exécution du programme. Contrairement aux conventions utilisées pour les (pseudo) chaînes du C, la notion de caractère de fin de chaîne n'existe plus et ce caractère de code nul peut apparaître au sein de la chaîne, éventuellement à plusieurs reprises. Un tel objet ressemble donc à un conteneur de type vector< char> et il possède d'ailleurs un certain nombre de fonctionnalités communes :

- l'accès aux éléments existants peut se faire avec l'opérateur [] ou avec la fonction membre at ; comme avec les vecteurs ou les tableaux usuels, le premier caractère correspond à l'indice 0 ;
- il possède une taille courante fournie par la fonction membre size() ;
- son emplacement est réservé sous forme d'un seul bloc de mémoire (ou, du moins, tout se passe comme si cela était le cas) ; la fonction capacity fournit le nombre maximal de caractères qu'on pourra y introduire, sans qu'il soit besoin de procéder à une nouvelle allocation mémoire ; on peut recourir aux fonctions reserve et resize ;

- on dispose des itérateurs à accès direct iterator et reverse_iterator, ainsi que des valeurs particulières begin(), end(), rbegin(), rend().

2. CONSTRUCTION

La classe string dispose de beaucoup de constructeurs ; certains correspondent aux constructeurs d'un vecteur :

```
string ch1 ;            /* construction d'une chaîne vide : ch1.size() == 0
*/
string ch2 (10, '*') ;  /* construction d'une chaîne de 10 caractères égaux à
'*' */
                        /*    ch2.size() == 10
*/
string ch3 (5, '\0') ; /* construction d'une chaîne de 5 caractères de code nul
*/
                       /*    ch2.size() == 5
*/
```

D'autres permettent d'initialiser une chaîne lors de sa construction, à partir de chaînes usuelles, constantes ou non :

```
string mess1 ("bonjour") ; /* construction d'une chaîne de 7 caractères :
bonjour */
char * adr = "salut" ;
string mess2 (adr) ;       /* construction d'une chaîne de 5 caractères : salut
*/
```

Bien entendu, on dispose d'un constructeur par recopie usuel :

```
string s1 ;
   .....
string s2(s1)  /* ou string s2 = s1 ;     construction de s2 par recopie de s1
*/
               /*  s2.size() == s1.size()
*/
```

Bien que d'un intérêt limité, on peut également construire une chaîne à partir d'une séquence de caractères, par exemple, si l est de type list< char> :

```
string chl (l.begin(), l.end()) ;  /* construction d'une chaîne en y recopiant
*/
                                   /* les caractères de la liste l
*/
```

3. OPERATIONS GLOBALES

On dispose tout naturellement des opérations globales déjà rencontrées pour les vecteurs, à savoir l'affectation, les fonctions assign et swap, ainsi que des comparaisons lexicographiques.

En outre, les opérateurs << et >> sont convenablement surdéfinis pour des objets de type string ; il en va de même de la fonction getline, rencontrée dans le paragraphe 2.3 du chapitre XVI dans le cas de chaînes usuelles.

4. CONCATENATION

L'opérateur + a été surdéfini de manière à permettre la concaténation :

- de deux objets de type string,
- d'un objet de type string avec une chaîne usuelle ou avec un caractère, et ceci dans n'importe quel ordre,

L'opérateur += est défini de façon concomitante.

Voici quelques exemples :

```
string ch1 ("bon") ;      /* ch1.size() == 3 */
string ch2 ("jour") ;     /* ch2.size() == 4 */
string ch3 ;              /* ch3.size() == 0 */
ch3 = ch1 + ch2 ;         /* ch3.size() == 7 ; ch3 contient la chaîne "bonjour"
*/
ch3 = ch1 + ' ' ;         /* ch3.size() == 4 */
ch3 += ch2 ;              /* ch3.size() == 8 ; ch3 contient la chaîne "bon jour"
*/
ch3 += " monsieur"        /* ch3 contient la chaîne "bon jour monsieur"
*/
```

On notera cependant qu'il n'est pas possible de concaténer deux chaînes usuelles ou une chaîne usuelle et un caractère :

```
char c1, c2 ;
ch3 = ch1 + c1 + ch2 + c2 ;       /* correct */
ch3 = ch1 + c1 + c2 ;             /* incorrect ; mais on peut toujours faire :
*/
                                  /*      ch3 = ch1 + c1 ; ch3 += c2 ;
*/
```

5. RECHERCHE DANS UNE CHAINE

Ces fonctions permettent de retrouver la première ou la dernière occurrence d'une chaîne ou d'un caractère donnés, d'un caractère appartenant à une suite de caractères donnés, d'un caractère n'appartenant pas à une suite de caractères donnés.

Lorsqu'une telle chaîne ou un tel caractère a été localisé, on obtient en retour l'indice correspondant au premier caractère concerné ; si la recherche n'aboutit pas, on obtient une valeur d'indice en dehors des limites permises pour la chaîne, ce qui rend quelque peu difficile l'examen de sa valeur.

5.1 Recherche d'une chaîne ou d'un caractère

La fonction membre find permet de rechercher, dans une chaîne donnée, la première occurrence :

- d'une autre chaîne (on parle souvent de sous-chaîne) fournie soit par un objet de type string, soit par une chaîne usuelle,

- d'un caractère donné.

Par défaut, la recherche commence au début de la chaîne, mais on peut la faire débuter à un caractère de rang donné.

Voici quelques exemples :

```
string ch = "anticonstitutionnellement" ;
string mot ("on");
char * ad = "ti" ;
int i ;
i = ch.find ("elle")  ;     /* i == 17               */
i = ch.find ("elles") ;     /* i <0 ou i > ch.size() */
i = ch.find (mot) ;         /* i == 5                */
i = ch.find (ad) ;          /* i == 2                */
i = ch.find ('n') ;         /* i == 1                */
i = ch.find ('n', 5) ;      /* i == 6  , car ici, la recherche débute à ch[5] */
i = ch.find ('p') ;         /* i <0 ou i > ch.size() */
```

De manière semblable, la fonction rfind permet de rechercher la dernière occurrence d'une autre chaîne ou d'un caractère.

```
string ch = "anticonstitutionnellement" ;
string mot ("on");
char * ad = "ti" ;
int i ;
i = ch.rfind ("elle")  ;     /* i == 17 */
i = ch.rfind ("elles") ;     /* i <0 ou i > ch.size() */
i = ch.rfind (mot) ;         /* i == 14 */
i = ch.rfind (ad) ;          /* i == 12 */
i = ch.rfind ('n') ;         /* i == 23 */
i = ch.rfind ('n', 18) ;     /* i == 16 */
```

5.2 Recherche d'un caractère présent ou absent d'une suite

La fonction find_first_of recherche la première occurrence de l'un des caractères d'une autre chaîne (string ou usuelle), tandis que find_last_of en recherche la dernière occurrence. La fonction find_first_not_of recherche la première occurrence d'un caractère n'appartenant pas à une autre chaîne, tandis que find_last_not_of en recherche la dernière. Voici quelques exemples :

```
string ch = "anticonstitutionnellement" ;
char * ad = "oie" ;
int i ;
i = ch.find_first_of ("aeiou")  ;       /* i == 0  */
i = ch.find_first_not_of ("aeiou")  ;   /* i == 1  */
i = ch.find_first_of ("aeiou", 6)  ;    /* i == 9  */
i = ch.find_first_not_of ("aeiou", 6)   /* i == 6  */
i = ch.find_first_of (ad) ;             /* i == 3  */
i = ch.find_last_of ("aeiou")  ;        /* i == 22 */
i = ch.find_last_not_of ("aeiou")  ;    /* i == 24 */
i = ch.find_last_of ("aeiou", 6)  ;     /* i == 5  */
i = ch.find_last_not_of ("aeiou", 6)    /* i == 6  */
i = ch.find_last_of (ad) ;              /* i == 22 */
```

6. INSERTIONS, SUPPRESSIONS ET REMPLACEMENTS

Ces possibilités sont relativement classiques, mais elles se recoupent partiellement, dans la mesure où l'on peut :

- d'une part utiliser, non seulement des objets de type string, mais aussi des chaînes usuelles (char *) ou des caractères,
- d'autre part définir une sous-chaîne, soit par indice, soit par itérateur, cette dernière possibilité n'étant cependant pas offerte systématiquement.

6.1 Insertions

La fonction insert permet d'insérer :

- à une position donnée, définie par un indice :
  - une autre chaîne (objet de type string) ou une partie de chaîne définie par un indice de début et une éventuelle longueur,
  - une chaîne usuelle (type char *) ou une partie de chaîne usuelle définie par une longueur,
  - une ou plusieurs fois un caractère donné ;
- à une position donnée définie par un itérateur :
  - une séquence d'éléments de type char, définie par un itérateur de début et un itérateur de fin,
  - une ou plusieurs fois un caractère donné.

Voici quelques exemples :

```
#include <iostream.h>
#include <string>
#include <list>
using namespace std ;
main()
{ string ch ("0123456") ;
  string voy ("aeiou") ;
  char t[] = {"778899"} ;
          /* insere le caractere a en ch.begin()+1 */
  ch.insert (ch.begin()+1, 'a') ;    cout << ch << "\n" ;
          /* insere le caractere b en position d'indice 4   */
  ch.insert (4, 'b') ;               cout << ch << "\n" ;
          /* insere 3 fois le caractere x en fin de ch  */
  ch.insert (ch.end(), 3, 'x') ;     cout << ch << "\n" ;
          /* insere 3 fois le caractere x en position d'indice 6 */
  ch.insert (6, 3, 'x') ;            cout << ch << "\n" ;
          /* insere la chaine voy en position 0 */
  ch.insert (0, voy) ;               cout << ch << "\n" ;
          /* insere, en position 3, la chaine voy, a partir de position
d'indice 2 */
  ch.insert (3, voy, 2) ;            cout << ch << "\n" ;
    /* insere en debut, la chaine voy, a partir de position d'indice 1,
longueur 3 */
```

```
  ch.insert (0, voy, 1, 3) ;          cout << ch << "\n" ;
          /* insertion d'une sequence */
  ch.insert (ch.begin()+2, t, t+6) ; cout << ch << "\n" ;
}
```

```
0a123456
0a12b3456
0a12b3456xxx
0a12b3xxx456xxx
aeiou0a12b3xxx456xxx
aeiiouou0a12b3xxx456xxx
eioaeiiouou0a12b3xxx456xxx
ei778899oaeiiouou0a12b3xxx456xxx
```

Exemple d'insertions dans une chaîne

6.2 Suppressions

La fonction remove permet de supprimer :

- une partie d'une chaîne, définie soit par un itérateur de début et un itérateur de fin, soit par un indice de début et une longueur ;
- un caractère donné défini par un itérateur de début.

Voici quelques exemples :

```
#include <iostream.h>
#include <string>
#include <list>
using namespace std ;
main()
{ string ch ("0123456789"), ch_bis=ch ;
       /* supprime, a partir de position d'indice 3, pour une longueur de 2 */
  ch.remove (3, 2) ;                        cout << "A : " << ch << "\n" ;
  ch = ch_bis ;
       /* supprime, de begin()+3 à begin()+6 */
  ch.remove (ch.begin()+3, ch.begin()+6) ;  cout << "B : " << ch << "\n" ;
       /* supprime, a partir de position d'indice 3   */
  ch.remove (3) ;                           cout << "C : " << ch << "\n" ;
  ch = ch_bis ;
       /* supprime le caractere de position begin()+4 */
  ch.remove (ch.begin()+4) ;                cout << "D : "<< ch << "\n" ;
}
```

```
A : 01256789
```

```
B : 0126789
C : 012
D : 012356789
```

Exemples de suppressions dans une chaîne

6.3 Remplacements

La fonction replace permet de remplacer une partie d'une chaîne définie, soit par un indice et une longueur, soit par un intervalle d'itérateur, par :

- une autre chaîne (objet de type string),
- une partie d'une autre chaîne définie par un indice de début et, éventuellement, une longueur,
- une chaîne usuelle (type char *) ou une partie de longueur donnée,
- un certain nombre de fois un caractère donné.

En outre, on peut remplacer une partie d'une chaîne définie par un intervalle par une autre séquence d'éléments de type char, définie par un itérateur de début et un itérateur de fin.

Voici quelques exemples :

```
#include <iostream.h>
#include <string>
using namespace std ;
main()
{ string ch ("0123456") ;
  string voy ("aeiou") ;
  char t[] = {"+*-/=<>"} ;
  char * message = "hello" ;
     /* remplace, a partir de indice 2, sur longueur 3, par voy */
  ch.replace (2, 3, voy) ;                                  cout << ch <<
"\n" ;
     /* remplace, a partir de indice 0 sur longueur 1, par voy,  */
     /*  a partir de indice 2, longueur 3                        */
  ch.replace (0, 1, voy, 1, 2) ;                             cout << ch
<< "\n" ;
     /* remplace, a partir de indice 1 sur longueur 2, par 8 fois '*' */
  ch.replace (1, 2, 8, '*') ;                                cout << ch
<< "\n" ;
     /* remplace, a partir de indice 1 sur longueur 2, par 5 fois '#' */
  ch.replace (1, 2, 5, '#') ;                                cout << ch
<< "\n" ;
     /* remplace, a partir de indice 2, sur longueur 4, par "xxxxxx" */
  ch.replace (2, 4, "xxxxxx" ) ;                             cout << ch
<< "\n" ;
     /* remplace les 7 derniers caracteres par les 3 premiers de message
*/
```

```
  ch.replace (ch.size()-7, ch.size(), message, 3) ;          cout << ch
<< "\n" ;
     /* remplace tous les caracteres, sauf le dernier, par (t, t+5) */
  ch.replace (ch.begin(), ch.begin()+ch.size()-1, t, t+5) ; cout << ch
<< "\n" ;
}
```

```
01aeiou56
ei1aeiou56
e********aeiou56
e##########aeiou56
e#xxxxxx******aeiou56
e#xxxxxx******hel
+*-/=l
```

Exemples de remplacements dans une chaîne

XXIII. LES OUTILS NUMÉRIQUES

La bibliothèque standard offre quelques patrons de classe destinés à faciliter les opérations mathématiques usuelles sur les nombres complexes et sur les vecteurs, de manière à munir C++ de possibilités voisines de celles de Fortran 90 et à favoriser son utilisation sur des calculateurs vectoriels ou parallèles. Il s'agit essentiellement :

- de la classe complex
- de la classe val_array et de classes assimilées.

1. LA CLASSE COMPLEX

Le patron de classe complexe offre de très riches outils de manipulation des nombres complexes. Il peut être paramétré par n'importe quel type flottant, float, double ou long double. Il comporte :

- les opérations arithmétiques usuelles : +, -, *, /,
- l'affectation (ordinaire ou composée comme +=, -=...),
- les fonctions de base : abs, arg, norm, real, imag,
- les fonctions "transcendantes" :
  - cos, sin, tan,
  - acos, asin, atan,
  - cosh, sinh, tanh,
  - exp, log.

Voici un exemple d'utilisation :

```
#include <complex>
using namespace std ;
main()
{ complex<float> z1(1, 2), z2(2, 5), z, zr ;
  cout << "z1 : " << z1 << "  z2 : " << z2 << "\n" ;
  cout << "Re(z1) : " << real(z1) << "  Im(z1) : " << imag(z1) << "\n" ;
  cout << "z1 + z2 : " << (z1+z2) << "  z1*z2 : " << (z1*z2)
       << "  z1/z2 : " << (z1/z2) << "\n" ;
  complex<double> i(0, 1) ;   // on definit la constante i
  z = 1+i ;
```

```
  zr = exp(z) ;
  cout << "exp(1+i) : " << zr << "  exp(i) : " << exp(i) << "\n" ;
  zr = log(i) ;
  cout << "log(i) : " << zr << "\n" ;
  double rho, theta, norme ;
  rho = abs(z) ; theta = arg(z) ; norme = norm(z) ;
  cout << "abs(1+i) : " << rho << "  arg(1+i) : " << theta
       << "  norm(1+i) : " << norme << "\n" ;
  double pi = 3.1415926535 ;
  cout << "cos(i) : " << cos(i) << "  sinh(pi*i): " << sinh(pi*i)
       << "  cosh(pi*i) : " << cosh(pi*i) << "\n" ;
}
```

```
z1 : (1,2)  z2 : (2,5)
Re(z1) : 1  Im(z1) : 2
z1 + z2 : (3,7)  z1*z2 : (-8,9)  z1/z2 : (0.413793,-0.0344828)
exp(1+i) : (1.46869,2.28736)  exp(i) : (0.540302,0.841471)
log(i) : (0,1.5708)
abs(1+i) : 1.41421  arg(1+i) : 0.785398  norm(1+i) : 2
cos(i) : (1.54308,0)  sinh(pi*i): (0,8.97932e-11)  cosh(pi*i) : (-1,0)
```

Exemples d'utilisation de nombres complexes

2. LA CLASSE VALARRAY

Ces différentes classes sont indépendantes des conteneurs décrits dans les chapitres précédents. Par ailleurs, elles ne sont actuellement pas connues par toutes les implémentations de C++ ; nous n'en donnerons qu'un bref aperçu.

La classe valarray est particulièrement adaptée aux tableaux numériques, c'est-à-dire dont les éléments sont d'un type de base. Voici quelques exemples de construction :

```
valarray<int>   vi (10) ;        /* tableau de 10 int                        */
valarray<float> vf (0.1, 20) ;   /* tableau de 20 float initialisés à 0.1    */
double t [] = { 1.25, 3.5, 0, 1.5 } ;
valarray<double> vd (t, t+4) ;   /* tableau de 4 double, initialisé avec les */
                                 /*   valeurs de t                           */
```

La classe valarray permet d'effectuer des opérations usuelles de calcul vectoriel en généralisant le rôle de tous les opérateurs et fonctions numériques : un opérateur unaire appliqué à un tableau fournit en résultat le tableau obtenu en appliquant cet opérateur à chacun de ses éléments ; un opérateur binaire appliqué à deux tableaux de même taille fournit en résultat le tableau obtenu en appliquant cet opérateur à chacun des éléments de même rang. Par exemple :

```
valarray<float> v1(5), v2(5), v3(5) ;
   .....
v3 = -v1 ;       /* v3[i] = -v1[i] pour i de 0 à 4 */
v3 = cos(v1) ;   /* v3[i] = cos(v1[i]) pour i de 0 à 4 */
v3 = v1 + v2 ;   /* v3[i] = v2[i] + v1[i] pour i de 0 à 4 */
v3 = v1*v2 + exp(v1) ;  /* v3[i] = v1[i]*v2[i] + exp(v1[i]) pour i de 0 à 4 */
```

Elle comporte également des opérateurs de comparaison (==, !=, <, <=...) qui s'appliquent à deux opérandes (de type valarray) de même nombre d'éléments et qui fournissent en résultat un tableau de booléens :

```
int dim = ... ;
valarray<float> v1(dim), v2(dim) ;
valarray<bool> egal(dim), inf(dim) ;
   .....
egal = (v1 == v2) ;  /* egal[i] = (v1[i] == v2[i]) pour i de 0 à dim-1 */
inf  = (v1 <  v2) ;  /* inf[i]  = (v1[i] <  v2[i]) pour i de 0 à dim-1 */
```

La classe mask_array permet de définir des "masques" (tableaux de booléens) utilisables dans la plupart des opérations appliquées aux tableaux[1].

Enfin, il est possible de définir des "sections" de tableaux ; on nomme ainsi un sous-ensemble des éléments d'un tableau sur lequel on peut travailler comme s'il s'agissait d'un tableau. La classe slice_array permet de définir des sections régulières, dans lesquelles on considère des éléments répartis de façon régulière ; elles sont caractérisées par un indice de début, un pas et un nombre d'éléments. La classe indirect_array permet de définir des sections irrégulières caractérisées par des vecteurs d'indices, c'est-à-dire qu'on précise l'emplacement de chaque élément. De plus, la classe gslice_array permet de définir des sections à plusieurs niveaux facilitant la manipulation de tableaux à plusieurs indices.

1 - On trouve des possibilités semblables en Fortran 90, avec l'instruction where.

ANNEXE A : RÈGLES DE MISE EN CORRESPONDANCE D'ARGUMENTS

Voici l'ensemble des règles présidant à la mise en correspondance d'arguments lors de l'appel d'une fonction surdéfinie ou d'un opérateur.

Nous décrivons tout d'abord la démarche employée pour les fonctions à un seul argument avant de voir comment elle se généralise aux fonctions à plusieurs arguments.

Notez bien que, comme nous l'avons signalé dans les chapitres correspondants, ces règles ne s'appliquent pas intégralement à l'instanciation d'une fonction patron.

1. CAS DES FONCTIONS À UN ARGUMENT

1.1 Recherche d'une correspondance exacte

Dans la recherche d'une correspondance exacte :

- On distingue bien les différents types entiers (char, short, int et long) avec leur attribut de signe ainsi que les différents types flottants (float, double et long double). Notez que, assez curieusement, char est à la fois différent de signed char et de unsigned char (alors que, dans une implémentation donnée[1], char est équivalent à l'un de ces deux types !).
- On ne tient pas compte des éventuels qualificatifs volatile et const, avec cependant deux exceptions pour const :
  - Depuis la version 2.0, on distingue un pointeur de type t * (t étant un type quelconque) d'un pointeur de type const t *, c'est-à-dire un pointeur sur une valeur constante de type t.

    Plus précisément, il peut exister deux fonctions, l'une pour le type t *, l'autre pour le type const t *. La présence ou l'absence du qualificatif const permettra de choisir la "bonne" fonction.

    S'il n'existe qu'une seule de ces deux fonctions correspondant au type const t *, t * constitue quand même une correspondance exacte pour const t * (là encore, ceci se justifie par le fait que le traitement prévu pour quelque chose de constant peut s'appliquer à quelque chose de non constant). En revanche, s'il n'existe qu'une fonction correspondant au type t *, const t * ne constitue pas une correspondance exacte pour ce type t * (ce qui signifie qu'on ne pourra pas appliquer à quelque chose de constant le traitement prévu pour quelque chose de non constant).

---

1 - Du moins, pour des options de compilation données.

- Depuis la version 3, on distingue le type t & (t étant un type quelconque et & désignant un transfert par référence) du type const t &. Le raisonnement précédent s'applique en remplaçant simplement t * par t &[2].

Si une fonction réalise une correspondance exacte, la recherche s'arrête là et la fonction trouvée est appelée. Notez qu'à ce niveau cette fonction est obligatoirement unique ; en effet, dans le cas contraire, les déclarations des différentes fonctions auraient été rejetées lors de leur compilation (par exemple, vous ne pourrez jamais définir f(int) et f(const int)).

1.2 Promotions numériques[3]

Si la recherche précédente n'a pas abouti, on effectue alors une nouvelle recherche, en faisant intervenir les conversions suivantes :

char, signed char, unsigned char, short -> int

unsigned short -> int ou unsigned int[4]

enum -> int

float -> double

Ici encore, si une fonction est trouvée, elle ne peut être qu'unique.

1.3 Conversions standard

Si la recherche n'a toujours pas abouti, on fait alors intervenir les conversions standard suivantes :

- type numérique en un autre type numérique (y compris des conversions "dégradantes"[5] : ainsi, un float conviendra là où un int est attendu),
- enum en un autre type numérique,
- 0 -> numérique,
- 0 -> pointeur quelconque,
- pointeur quelconque -> void *[6],
- pointeur sur une classe dérivée -> pointeur sur une classe de base.

Cette fois, il est possible que plusieurs fonctions conviennent. Il y a alors ambiguïté, excepté quelques situations :

- la conversion d'un pointeur sur une classe dérivée en un pointeur sur une classe de base est préférée à la conversion en void *,
- si C dérive de B et que B dérive de A, la conversion C * en B * est préférée à la conversion en A * ; il en va de même pour la conversion C & en B & qui est préférée à la conversion en A &.

---

2 - En toute rigueur, on distingue également volatile t* de t* et volatile t & de t &.

3 - Elles n'existaient pas dans les versions antérieures à la version 2.0.

4 - Suivant que, dans l'implémentation concernée, un int suffit ou ne suffit pas à accueillir un unsigned short (il ne le peut pas lorsque short et int correspondent au même nombre de bits).

5 - Dans les versions antérieures à la 2.0, on se limitait aux conversions "non dégradantes".

6 - La conversion inverse n'est pas prévue. Ceci est cohérent avec le fait qu'en C++, contrairement à ce qui se passe en C ANSI, un pointeur de type void * ne peut pas être affecté à un pointeur quelconque.

1.4 Conversions définies par l'utilisateur

Si aucune fonction ne convient, on fera intervenir les "conversions définies par l'utilisateur" (C.D.U.).

Une seule C.D.U. pourra intervenir ; en revanche, elle pourra être associée à d'autres conversions. Toutefois, lorsqu'une chaîne de conversions peut être simplifiée en une chaîne plus courte, seule cette dernière est considérée ; par exemple, dans char -> int -> float et char -> float , on ne considère que char -> float. Ici encore, si plusieurs combinaisons de conversions existent (après les éventuelles simplifications évoquées), le compilateur refusera l'appel à cause de son ambiguïté.

1.5 Fonctions à arguments variables

Lorsqu'une fonction a prévu des arguments de types quelconques (notation "..."), n'importe quel type d'argument effectif convient.

Notez bien que cette possibilité n'est examinée qu'en dernier. Cette remarque prendra tout son intérêt dans le cas de fonctions à plusieurs arguments.

1.6 Exception

Lorsqu'une fonction possède un argument muet qui est une référence (autrement dit qu'il est du type T&), la correspondance d'argument doit permettre à la fonction de travailler "directement" avec la variable concernée, donc éventuellement de la modifier. Cela implique obligatoirement des restrictions sur le type de l'argument effectif correspondant : il doit s'agir d'une lvalue de type T.

Une seule exception a lieu lorsque l'argument muet possède l'attribut const car, dans ce cas, on peut mettre à disposition de la fonction la référence à une valeur temporaire obtenue à partir de la valeur de l'argument effectif (avec d'éventuelles conversions).

2. CAS DES FONCTIONS À PLUSIEURS ARGUMENTS

Le compilateur recherche une fonction "meilleure" que toutes les autres. Pour ce faire, il applique les règles de recherche précédentes à chacun des arguments. Cela l'amène à sélectionner, pour chaque argument, une ou plusieurs fonctions réalisant la meilleure correspondance ; cette fois, il peut y en avoir plusieurs car la détermination finale de la "bonne fonction" n'est pas encore faite (toutefois, si aucune fonction n'est sélectionnée pour un argument donné, on est déjà sûr qu'aucune fonction ne conviendra). Ensuite de quoi, le compilateur détermine, pour toutes les fonctions ainsi sélectionnées, celle, si elle existe et si elle est unique, qui réalise la meilleure correspondance, c'est-à-dire celle pour laquelle la correspondance de chaque argument est égale ou supérieure à celle des autres[7].

Remarque :

7 - En fait, cela revient à dire, en terme ensemblistes, qu'on considère l'intersection des différents ensembles formés des fonctions réalisant la meilleure correspondance pour chaque argument. Cette intersection doit comporter exactement un élément.

Les fonctions comportant un ou plusieurs arguments par défaut sont traitées comme si plusieurs fonctions différentes avaient été définies avec un nombre croissant d'arguments.

3. CAS DES FONCTIONS MEMBRE

Un appel de fonction membre (non statique[8]) peut être considéré comme un appel d'une fonction ordinaire, auquel s'ajoute un argument effectif ayant le type de l'objet ayant effectué l'appel. Toutefois, cet argument n'est **pas du tout soumis aux règles de correspondance** dont nous parlons ici ; en effet, c'est son type qui détermine la fonction membre à appeler (avec éventuellement prise en compte d'un mécanisme d'héritage).

Les seules "nuances" qui puissent intervenir concernent les attribut const et volatile. En effet, il est possible de distinguer une fonction membre agissant sur des objets constants d'une fonction membre agissant sur des objets non constants. Une fonction membre constante peut toujours agir sur des objets non constants ; la réciproque est bien sûr fausse. La même remarque s'applique à l'attribut volatile.

8 - Une fonction membre statique ne comporte aucun argument implicite de type classe.

ANNEXE B : LES INCOMPATIBILITÉS ENTRE C ET C++

Nous récapitulons ici l'ensemble des incompatiblités existant entre le C ANSI et le C++ (dans ce sens), c'est-à-dire les différents points acceptés par le C ANSI et refusés par le C++. Notez que les cinq premiers points ont été exposés en détail dans le chapitre II, le dernier l'a été dans le chapitre IV. Les autres correspondent à des usages assez peu fréquents.

1. Prototypes

En C++, toute fonction non définie préalablement dans un fichier source où elle est utilisée doit faire l'objet d'une déclaration sous forme d'un prototype.

2. Fonctions sans arguments

En C++, une fonction sans arguments se définit (en-tête) et se déclare (prototype) en fournissant une "liste vide" d'arguments comme dans :

```
float fct () ;
```

3. Fonctions sans valeur de retour

En C++, une fonction sans valeur de retour se définit (en-tête) et se déclare (prototype) **obligatoirement** à l'aide du mot void comme dans :

```
void fct (int, double) ;
```

4. Le qualificatif const

En C++, un symbole accompagné, dans sa déclaration, du qualificatif const a une portée limitée au fichier source concerné, alors qu'en C ANSI il est considéré comme un symbole externe. De plus, en C++, un tel symbole peut intervenir dans une expression constante (il ne s'agit toutefois plus d'une incompatibilité mais d'une liberté offerte par C++).

5. Les pointeurs de type void *

En C++, un pointeur de type void * ne peut pas être converti implicitement en un pointeur d'un autre type.

6. Mots clés

C++ possède, par rapport à C, les mots clés supplémentaires suivants[1] :

[1] - Le mot clé overload a existé dans les versions antérieures à la 2.0. S'il reste reconnu de certaines implémentations, en étant alors sans effet, il ne figure cependant pas dans la norme.

bool
catch
class
const_cast
delete
dynamic_cast
explicit
false
friend
inline
mutable
namespace
new
operator
private
protected
public
reinterpret_cast
static_cast
template
this
true
throw
try
typeid
typename
using
virtual

Voici la liste complète des mots clés de C++ ; ceux qui existent déjà en C sont écrits en romain tandis que ceux qui sont propres à C++ sont écrits en italique ; à simple titre indicatif, les mots clés introduits tardivement par la norme ANSI sont écrits en gras (et en italique).

asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while

7. Les constantes de type caractère

En C++ (depuis la version 2.0), une constante caractère telle que 'a', 'z' ou '\n' est de type char, alors qu'elle est implicitement convertie en nt en C ANSI (et dans les versions de C++ antérieures à la 2.0). Cela permet notamment de distinguer, par exemple, les deux fonctions suivantes :

```
fct (char) ;

fct (int) ;
```

C'est ainsi que l'opérateur << de la classe ostream peut fonctionner correctement avec des caractères (dans les versions antérieures à la 2.0, on obtient le code numérique du caractère).

Notez bien qu'une expression telle que :

```
sizeof ('a')
```

vaut 1 en C++ (depuis la version 2.0), alors qu'elle vaut davantage (généralement 2 ou 4) en C ou dans les versions de C++ antérieures à la 2.0.

8. Les définitions multiples

En C ANSI, il est permis de trouver plusieurs déclarations d'une même variable dans un fichier source. Par exemple, avec :

```
int n ;
 .....
int n ;
```

C considère que la première instruction est une simple déclaration, tandis que la seconde est une définition ; c'est cette dernière qui provoque la réservation de l'emplacement mémoire pour n.

En C++, **cela est interdit**. La raison principale vient de ce que, dans le cas où de telles déclarations porteraient sur des objets, par exemple dans :

```
point a ;
 .....
point a ;
 .....
```

il faudrait que le compilateur distingue déclaration et définition de l'objet point et qu'il prévoie de n'appeler le constructeur que dans le second cas. Cela aurait été particulièrement dangereux, d'où l'interdiction adoptée.

9. L'instruction goto

En C++, une instruction goto ne peut pas faire sauter une déclaration comportant un "initialiseur" (par exemple int n = 2), sauf si cette déclaration figure dans un bloc et que ce bloc est sauté complètement.

10. Les énumérations

En C++, les éléments d'une énumération (mot clé enum) ont une portée limitée à l'espace de visibilité dans lequel ils sont définis. Par exemple, avec :

```
struct chose
{  enum (rouge = 1, bleu, vert) ;
   .....
} ;
```

les symboles rouge, bleu et vert ne peuvent pas être employés en dehors d'un objet de type chose. Ils peuvent éventuellement être redéfinis avec une signification différente. En C, ces symboles sont accessibles de toute la partie du fichier source suivant leur déclaration et il n'est alors plus possible de les redéfinir.

11. Initialisation de tableaux de caractères

En C++, l'initialisation de tableaux de caractères par une chaîne de même longueur n'est pas possible. Par exemple, l'instruction :

```
char t[5] = "hello" ;
```

provoquera une erreur, due à ce que t n'a pas une dimension suffisante pour recevoir le caractère (\0) de fin de chaîne.

En C ANSI, cette même déclaration serait acceptée et le tableau t se verrait simplement initialisé avec les 5 caractères h, e, l, l et o (sans caractère de fin de chaîne).

Notez que l'instruction :

```
char t[] = "hello" ;
```

convient indifféremment en C et en C++ et qu'elle réserve dans les deux cas un tableau de 6 caractères : h, e, l, l, o et \0.

12. Les noms de fonctions

En C++, depuis la version 2.0, le compilateur attribue à toutes les fonctions un "nom externe" basé d'une façon déterministe :

- sur son nom "interne",
- sur la nature de ses arguments.

Si l'on veut obtenir les mêmes noms de fonction qu'en C, on peut faire appel au mot clé extern. Pour plus de détails, voyez le paragraphe 5.3 du chapitre IV.

ANNEXE C : OPÉRATEURS DE CAST ET IDENTIFICATION DE TYPE À L'ÉXÉCUTION

Cette annexe présente quelques points qui ont été introduits assez tardivement dans la norme ANSI et qui risquent de ne pas exister dans certaines implémentations. Il s'agit essentiellement :

- de possibilités d'identification dynamique du type des objets au moment de l'exécution,
- de nouveaux opérateurs de "cast",
- de possibilités de "cast" dynamiques,

1. IDENTIFICATION DE TYPE À L'EXÉCUTION

La norme ANSI prévoit dans C++ un mécanisme permettant de connaître (identifier et comparer), lors de l'exécution du programme, le type d'une variable, d'une expression ou d'un objet.

Bien entendu, cela ne présente guère d'intérêt si un tel type est défini lors de la compilation. Ainsi, avec :

```
int n ;
float x ;
```

il ne sera guère intéressant de savoir que le type de n ou celui de x peuvent être connus ou encore que n et x sont d'un type différent. La même remarque s'appliquerait à des objets d'un type classe.

En fait, cette possibilité a été essentiellement introduite pour être utilisée dans les situations de polymorphisme, comme nous l'avons décrit dans le chapitre relatif aux fonctions virtuelles.

Plus précisément, il est possible, lors de l'exécution, de connaître le **véritable type d'un objet désigné par un pointeur ou par une référence.**

Pour ce faire, il existe un opérateur nommé typeid à un opérande fournissant en résultat un objet de type prédéfini Type_info. Cette classe contient la fonction membre name(), laquelle fournit une chaîne de caractères représentant le nom du type ; ce nom n'est pas imposé par la norme ; il peut donc dépendre de l'implémentation mais on est sûr que deux types différents n'auront jamais le même nom.

De plus, la classe dispose de deux opérateurs binaires == et != qui permettent de comparer deux types.

1.1 Utilisation du champ name de Type_info

Voici un premier exemple inspiré du programme utilisé pour illustrer le mécanisme des fonctions virtuelles (paragraphe 2 du chapitre XV) ; il montre l'intérêt que présente typeid lorsqu'on l'applique dans un contexte de polymorphisme.

```
#include <iostream.h>
#include <typeinfo.h>     // pour typeid
class point
{ public :
     virtual void affiche ()
      { }          // ici vide - utile pour le polymorphisme
} ;

class pointcol : public point
{ public :
     void affiche ()
      { }          // ici vide
} ;
main()
{ point p ; pointcol pc ;
  point * adp ;
  adp = &p ;
  cout << "type de adp  : " << typeid (adp).name()  << "\n" ;
  cout << "type de *adp : " << typeid (*adp).name() << "\n"  ;
  adp = &pc ;
  cout << "type de adp  : " << typeid (adp).name()  << "\n" ;
  cout << "type de *adp : " << typeid (*adp).name() << "\n"  ;
}
```

```
type de adp  : point *
type de *adp : point
type de adp  : point  *
type de *adp : pointcol
```

Exemple d'utilisation de l'opérateur typeid

On notera bien que, pour typeid, le type du pointeur adp reste bien point *, mais qu'en revanche, le type de l'objet pointé (*adp) quant à lui, est bien déterminé par la nature exacte de l'objet pointé.

Remarques :

1) Rappelons que la norme n'impose pas le nom exact que doit fournir cet opérateur ; on n'est donc pas assuré que le nom de type sera toujours point, point *, pointcol * comme ici.

2) Ici, les méthodes affiche ont été prévues vides ; elles ne servent en fait qu'à assurer le polymorphisme. En son absence, l'opérateur typeid se contenterait de fournir comme type d'un objet pointé celui défini par le type (statique) du pointeur.

3) Notez bien que si les fonctions virtuelles permettaient d'aboutir à un certain typage dynamique, ce dernier ne concernait que les fonctions membre appelées : on appelait bien la fonction membre correspondant au type de l'objet pointé, mais on ne disposait, en revanche, d'aucun moyen permettant de connaître le type de cet objet pointé. Ou, pour exprimer les choses d'une autre manière, on obtenait le comportement correspondant au type de l'objet pointé, mais on ne pouvait pas en connaître explicitement le type.

1.2 Utilisation des opérateurs de comparaison de Type_info

Voici, toujours inspiré du programme relatif aux fonctions virtuelles, un exemple montrant l'usage de l'opérateur ==.

```
#include <iostream.h>
#include <typeinfo.h>     // pour typeid
class point
{ public :
     virtual void affiche ()
      { }        // ici vide - utile pour le polymorphisme
} ;
class pointcol : public point
{ public :
     void affiche ()
      { }        // ici vide
} ;
main()
{ point p1, p2 ;
  pointcol pc ;
  point * adp1, * adp2 ;
  adp1 = &p1 ; adp2 = &p2 ;
  cout << "En A : les objets pointes par adp1 et adp2 sont " ;
  if (typeid(*adp1) == typeid (*adp2)) cout << "meme type\n" ;
                                  else cout << "type different\n" ;
  adp1 = &p1 ; adp2 = &pc ;
  cout << "En B : les objets pointes par adp1 et adp2 sont " ;
  if (typeid(*adp1) == typeid (*adp2)) cout << "meme type\n" ;
                                  else cout << "type different\n" ;
}
```

```
En A : les objets pointes par adp1 et adp2 sont de meme type
En B : les objets pointes par adp1 et adp2 sont de type different
```

Exemple de comparaison de types dynamiques avec l'opérateur typeid

1.3 Exemple avec des références

Voici un dernier exemple où l'on applique l'opérateur== de typeid à des références. On voit qu'on dispose ainsi d'un moyen de s'assurer dynamiquement (au moment de l'exécution) de l'identité de type de deux objets reçus en argument d'une fonction.

```
#include <iostream.h>
#include <typeinfo.h>     // pour typeid
class point
{ public :
      virtual void affiche ()
        { }          // ici vide
} ;
class pointcol : public point
{ public :
      void affiche ()
        { }          // ici vide
} ;
void fct (point & a, point & b)
{ if (typeid(a) == typeid(b))
              cout << "reference a des objets de meme type \n" ;
        else cout << "reference a des objets de type different \n" ;
}
main()
{ point p ;
  pointcol pc1, pc2 ;
  cout << "Appel A : " ; fct (p, pc1) ;
  cout << "Appel B : " ; fct (pc1, pc2) ;
}
```

```
Appel A : reference a des objets de meme type
Appel B : reference a des objets de type different
```

Exemple d'utilisation de typeid avec des références

2. LES NOUVEAUX OPÉRATEURS DE "CAST"

En C++, comme en C, il est possible de réaliser des conversions explicites, à l'aide d'un opérateur de "cast". Les conversions acceptées comportent naturellement toutes les conversions implicites légales auxquelles s'ajoutent quelques autres pouvant être dégradantes ou dépendantes de l'implémentation.

La norme ANSI prévoit de conserver ces possibilités, tout en proposant de nouveaux opérateurs de "cast", plus évocateurs de la nature de la conversion et de sa portabilité éventuelle ; ils sont formés comme les opérateurs classiques à l'aide du type souhaité, lequel est complété d'un mot clé précisant le type de conversion :

- const_cast : pour ajouter ou supprimer à un type l'un des modificateurs const ou volatile,

- reinterpret_cast : pour les conversions dont le résultat dépend de l'implémentation ; typiquement, il s'agit des conversions d'entier vers pointeur et de pointeur vers entier.
- static_cast : pour les conversions indépendantes de l'implémentation. En fait, les conversions de pointeur vers pointeur entrent dans cette catégorie, malgé les différences qui peuvent apparaître liées aux contraintes d'alignement propres à chaque implémentation.

Voici quelques exemples commentés :

```
#include <iostream.h>
main ()
{
  int n = 12 ;
  const int * ad1 = &n ;
  int * ad2 ;

  ad2 = (int *) ad1 ;                    // ancienne forme conseillée
                                         //  (ad2 = ad1 serait rejetée)
  ad2 = const_cast <int*> (ad1) ;        // forme ANSI conseillée
  ad1 = ad2 ;                            // légale
  ad1 = (const int *) ad2 ;              // forme ancienne conseillée
  ad1 = const_cast <const int *> (ad2) ; // forme ANSI conseillée

  const int p = 12 ;
  const int const * ad3 = &p ;
  int * ad4 ;

  ad4 = (int *) ad3 ;                     // ancienne forme conseillée
                                          //  (ad4 = ad3 serait rejetée)
  ad4 = const_cast <int *> (ad3) ;        // forme ANSI conseillée

  ad3 = ad4 ;                                  // légale
  ad3 = (const int const *) ad4 ;              // ancienne forme conseillée
  ad3 = const_cast <const int const *> (ad4) ; // forme ANSI conseillée
}
```

Exemples d'utilisation de l'opérateur const_cast < ... >

```
#include <iostream.h>
main ()
{ long n ;
  int * adi ;
  adi = (int *) n ;                     // ancienne forme conseillée
                                        //   (adi = n  serait rejetée)
  adi = reinterpret_cast <int *> (n) ;  // forme ANSI conseillée

  n = (long) adi ;                      // ancienne forme conseillée
                                        //   (n = adi serait rejetée)
```

```
    n = reinterpret_cast <long> (adi) ;    // forme ANSI conseillée

    int p ;
    p = n ;                                // acceptée
    p = (int) n ;                          // ancienne forme conseillée
    p = static_cast <int> (n) ;            // forme ANSI conseillée
}
```

Exemples d'utilisation des opérateurs reinterpret_cast < ... > et static_cast

Remarque :

Ces nouveaux opérateurs n'apportent aucune possibilité de conversion supplémentaire. Il n'en ira pas de même de l'opérateur dynamic_cast, étudié ci-après.

3. LES "CAST" DYNAMIQUES

3.1 Introduction

Nous venons de voir comment les possibilités d'identification des types à l'exécution complètent le polymorphisme offert par les fonctions virtuelles en permettant d'identifier le type des objets pointés ou référencés.

Cependant, une lacune subsiste : on sait agir sur l'objet pointé en fonction de son type, on peut connaître le type exact de cet objet mais le type proprement dit des pointeurs utilisés dans ce polymorphisme reste celui défini à la compilation. Par exemple, si l'on sait que adp pointe sur un objet de type pointcol (dérivé de point), on pourrait souhaiter convertir sa valeur en un pointeur de type pointcol *.

Le projet de norme C++ prévoit cette possibilité, par le biais d'opérateurs dits "cast dynamiques". Ainsi, avec l'hypothèse précédente (on est sûr que adp pointe réellement sur un objet de type pointcol), on pourra écrire :

```
pointcol * adpc = dynamic_cast <pointcol *> (adp) ;
```

Bien entendu, en compilation, la seule vérification qui sera faite est que cette conversion est (peut-être) acceptable car l'objet pointé par adp est d'un type point ou dérivé et pointcol est lui-même dérivé de point. Mais, ce n'est qu'au moment de l'exécution qu'on saura si la conversion est réalisable ou non. Par exemple, si adp pointait sur un objet de type point, la conversion échouerait.

3.2 D'une manière générale

L'opérateur dynamic_cast aboutit si l'objet réellement pointé est, par rapport au type d'arrivée demandé, d'un type identique ou d'un type descendant (mais dans un contexte de polymorphisme, c'est-à-dire qu'il doit exister au moins une fonction virtuelle).

Lorsque l'opérateur n'aboutit pas :

- il fournit le pointeur NULL s'il s'agit d'une conversion de pointeur,
- il déclenche une exception bad_cast s'il s'agit d'une conversion de référence.

3.3 Exemple

Voici un exemple faisant intervenir une hiérarchie de trois classes dérivées les unes des autres.

```
#include <iostream.h>
#include <typeinfo.h>     // pour typeid
class A
{ public :
    virtual void affiche ()   // vide ici - utile pour le polymorphisme
     { }
} ;
class B : public A
{ public :
    void affiche ()
     { }
} ;
class C : public B
{ public :
    void affiche ()
     { }
} ;
main()
{ A a ; B b ; C c ;
  A * ada, * ada1 ;
  B * adb, * adb1 ;
  C * adc ;
  ada = &a ;    // ada de type A* pointe sur un A ;
                //  sa conversion dynamique en B* ne marche pas
  adb =  dynamic_cast <B *> (ada) ; cout << "dc <B*>(ada) " << adb << "\n" ;
  ada = &b ;    // ada de type A* pointe sur un B ;
                // sa conversion dynamique en B* marche
  adb =  dynamic_cast <B *> (ada) ; cout << "dc <B*> ada " << adb << "\n" ;
               // sa conversion dynamique en A* marche
  ada1 = dynamic_cast <A*> (ada) ; cout << "dc <A*> ada " << ada1 << "\n" ;
              // mais sa conversion dynamique en C* ne marche pas
  adc =  dynamic_cast <C *> (ada) ; cout << "dc <C*> ada " << adc << "\n" ;
  adb = &b ;    // adb de type B* pointe sur un B
                // sa conversion dynamique en A* marche
  ada1 = dynamic_cast <A *> (adb) ; cout << "dc <A*> adb " << ada1 << "\n" ;
                // sa conversion dynamique en B* marche
  adb1 = dynamic_cast <B *> (adb) ; cout << "dc <A*> adb1 " << adb1 << "\n" ;
                // mais sa conversion dynamique en C* ne marche pas
  adc =  dynamic_cast <C *> (adb) ; cout << "dc <C*> adb1 " << adc << "\n" ;
}
```

```
dc <B*>(ada) 0x00000000
dc <B*> ada 0x54820ffc
dc <A*> ada 0x54820ffc
dc <C*> ada 0x00000000
dc <A*> adb 0x54820ffc
dc <A*> adb1 0x54820ffc
dc <C*> adb1 0x00000000
```

Exemple d'utilisation de l'opérteur dynamic_cast

ANNEXE D : LES DIFFÉRENTES SORTES DE FONCTIONS EN C++

Nous vous fournissons ici la liste des différentes sortes de fonctions que l'on peut rencontrer en C++ en précisant, dans chaque cas, si elle peut être définie comme fonction membre ou amie, s'il existe une version par défaut, si elle est héritée et si elle peut être virtuelle.

Type de fonction	Membre ou amie	Version par défaut	Héritée	Peut être virtuelle
constructeur	membre	oui	non	non
destructeur	membre	oui	non	oui
conversion	membre	non	oui	oui
affectation	membre	oui	non	oui
()	membre	non	oui	oui
[]	membre	non	oui	oui
->	membre	non	oui	oui
new	membre statique	non	oui	oui
delete	membre statique	non	oui	oui
autre opérateur	l'un ou l'autre	non	oui	oui
autre fonction membre	membre	non	oui	oui
fonction amie	amie	non	non	non

ANNEXE E : COMPTAGE DE RÉFÉRENCES

Nous avons vu que, dès lors qu'un objet comporte une partie dynamique, il est nécessaire de procéder à des copies "profondes" plutôt qu'à des copies "superficielles" et ceci aussi bien dans le constructeur de recopie que dans l'opérateur d'affectation. Dans ce cas, on peut se demander s'il est vraiment nécessaire de dupliquer la partie dynamique de l'objet.

En fait, **si cette partie dynamique est toujours manipulée globalement**[1], il est possible d'éviter une telle duplication en faisant appel à la technique du "compteur de références". Elle consiste à compter, en permanence, le nombre de références à un emplacement dynamique, c'est-à-dire le nombre de pointeurs différents la désignant à un instant donné. Dans ces conditions, lorsqu'un objet est détruit, il suffit de n'en détruire la partie dynamique correspondante que si son compteur de références est nul pour éviter les risques de libération multiple que nous avons souvent évoqués.

Pour mettre en œuvre cette technique, deux points doivent être précisés.

a) L'emplacement du compteur de références

A priori, deux possibilités viennent à l'esprit : dans l'objet lui-même ou dans la partie dynamique associée à l'objet. La première solution n'est guère exploitable car elle obligerait d'une part à dupliquer ce compteur autant de fois qu'il y a d'objets pointant sur une même zone et, d'autre part, il serait très difficile d'effectuer la mise à jour des compteurs de tous les objets désignant la même zone. Manifestement donc, le compteur de référence doit être associé, non pas à un objet, mais à sa partie dynamique.

b) Les méthodes devant agir sur le compteur de références

Le compteur de références doit être mis à jour chaque fois que le nombre d'objets désignant l'emplacement correspondant risque d'être modifié. Cela concerne donc :

- le constructeur de recopie : il doit initialiser un nouvel objet pointant sur un emplacement déjà référencé et donc incrémenter son compteur de références,
- l'opérateur d'affectation ; une instruction telle que a = b doit :
  - décrémenter le compteur de références de l'emplacement référencé par a et procéder à sa libération lorsque le compteur est nul,
  - incrémenter le compteur de références de l'emplacement référencé par b.

Bien entendu, il est indispensable que le constructeur de recopie existe et que l'opérateur d'affectation soit surdéfini. Le non-respect de l'une de ces deux conditions et l'utilisation des méthodes par défaut qui en découle entraîneraient des recopies d'objets sans mise à jour des compteurs de références...

---

1 - Ce ne serait pas le cas par exemple pour une classe "vecteur dynamique" dans laquelle on aurait surdéfini l'opérateur [].

Nous vous proposons un "canevas général" applicable à toute classe de type X possédant une partie dynamique de type T. Ici, pour réaliser l'association de la partie dynamique et du compteur associé, nous utilisons une structure de nom partie_dyn. La partie dynamique de X sera gérée par un pointeur sur une structure de type partie_dyn.

```
// T désigne un type quelconque (éventuellement classe)

struct partie_dyn      // structure "de service" pour la partie dynamique de
l'objet
{  long nref ;           // compteur de référence associé
   T * adr ;             // pointeur sur partie dynamique (de type T)
} ;
class X
{   // membres donnée non dynamiques
    //   .....
   partie_dyn * adyn ;           // pointeur sur partie dynamique
   void decremente ()            // fonction "de service" - décrémente le
     { if (!--adyn->nref)        // compteur de référence et détruit
         {  delete adyn->adr ;   // la partie dynamique si nécessaire
            delete adyn ;
         }
     }
 public :
   X (  )                    // constructeur "usuel"
     { // construction partie non dynamique
     //   .....
     // construction partie dynamique
        adyn = new partie_dyn  ;
        adyn->adr = new T ;
        adyn->nref = 1 ;
     }
   X (X & x)            // constructeur de recopie
     { // recopie partie non dynamique
       //   .....
       // recopie partie dynamique
       adyn  = x.adyn ;
       adyn->nref++ ;               // incrémentation compteur références
     }
   ~X ()                   // destructeur
     { decremente () ;
     }
   X & operator = (X & x)    // surdéfinition opérateur affectation
     { if (this != &x)             // on ne fait rien pour a=a
       // traitement partie non dynamique
       //   .....
       // traitement partie dynamique
         { decremente () ;
           x.adyn->nref++ ;
           adyn = x.adyn ;
         }
       return * this ;
```

```
        }
} ;
```

Un canevas général pour le "comptage de références"

ANNEXE F : LES ALGORITHMES STANDARD

Cette annexe fournit le rôle exact des algorithmes proposés par la bibliothèque standard. Ils sont classés suivant les mêmes catégories que celles du chapitre XXI qui explique le fonctionnement de la plupart d'entre eux. La nature des itérateurs reçus en argument est précisée en utilisant les abréviations suivantes :

- Ie Itérateur d'entrée,
- Is Itérateur de sortie,
- Iu Itérateur unidirectionnel,
- Ib Itérateur bidirectionnel,
- Ia Itérateur à accès direct.

Nous indiquons la complexité de chaque algorithme, dans le cas où elle n'est pas triviale. Comme le fait la norme, nous l'exprimons en un nombre précis d'opérations (éventuellement sous forme d'un maximum), plutôt qu'avec la notation de Landau moins précise. Pour alléger le texte, nous avons convenu que lorsqu'une seule séquence est concernée, N désigne son nombre d'éléments ; lorsque deux séquences sont concernées, N1 désigne le nombre d'éléments de la première et N2 celui de la seconde. Dans quelques rares cas, d'autres notations seront nécessaires : elles seront alors explicitées dans le texte.

Notez que, par souci de simplicité, lorsqu'aucune ambiguïté n'existera, nous utiliserons souvent l'abus de langage qui consiste à parler des éléments d'un intervalle [début, fin) plutôt que des éléments désignés par cet intervalle. D'autre part, les prédicats ou fonctions de rappel prévus dans les algorithmes correspondent toujours à des objets fonction ; cela signifie qu'on peut recourir à des classes fonction prédéfinies, à ses propres classes fonction ou à des fonctions ordinaires.

1. ALGORITHMES D'INITIALISATION DE SEQUENCES EXISTANTES

FILL `void fill (Iu début, Iu fin, valeur)`

Place valeur dans l'intervalle [début, fin)

FILL_N `void fill_n (Is position, NbFois, valeur)`

Place valeur NbFois consécutives à partir de position ; les emplacements correspondants doivent exister.

COPY `Is copy (Ie début, Ie fin, Is position)`

Copie l'intervalle [début, fin), à partir de position ; les emplacements correspondants doivent exister ; la valeur de position (et seulement celle-ci) ne doit pas appartenir à l'intervalle

[début, fin) ; si tel est le cas, on peut toujours recourir à copy_backward ; renvoie un itérateur sur la fin de l'intervalle où s'est faite la copie.

COPY_BACKWARD `lb copy_backward (lb début, lb fin, lb position)`

Comme copy, copie l'intervalle [début, fin), en progressant du dernier élément vers le premier, à partir de position qui désigne donc l'emplacement de la première copie, mais aussi la fin de l'intervalle ; les emplacements correspondants doivent exister ; la valeur de position (et seulement celle-ci) ne doit pas appartenir à l'intervalle [début, fin) ; renvoie un itérateur sur le début de l'intervalle (dernière valeur copiée) où s'est faite la copie ; cet algorithme est surtout utile en remplacement de copy lorsque le début de l'intervalle d'arrivée appartient à l'intervalle de départ.

GENERATE `void generate (lu début, lu fin, fct_gen)`

Appelle, pour chacune des valeurs de l'intervalle [début, fin), la fonction fct_gen et affecte la valeur fournie à l'emplacement correspondant.

GENERATE_N `void generate_n (lu début, NbFois, fct_gen)`

Même chose que generate, mais l'intervalle est défini par sa position début et son nombre de valeurs NbFois (la fonction fct_gen est bien appelée NfFois).

SWAP_RANGES `lu swap_ranges (lu début_1, lu fin_1, lu début_2)`

Echange les éléments de l'intervalle [début, fin) avec l'intervalle de même taille commençant en début_2. Les deux intervalles ne doivent pas se chevaucher. Complexité : N échanges.

2. ALGORITHMES DE RECHERCHE

FIND `le find (le début, le fin, valeur)`

Fournit un itérateur sur le premier élément de l'intervalle [début, fin) égal à valeur (au sens de ==) s'il existe, la valeur fin sinon ; (attention, il ne s'agit pas nécessairement de end()). Complexité : au maximum N comparaisons d'égalité.

FIND_IF `le find_if (le début, le fin, prédicat_u)`

Fournit un itérateur sur le premier élément de l'intervalle [début, fin) satisfaisant au prédicat unaire prédicat_u spécifié, s'il existe, la valeur fin sinon ; (attention, il ne s'agit pas nécessairement de end()). Complexité : au maximum N appels du prédicat.

FIND_END `lu find_end (lu début_1, lu fin_1, lu début_2, lu fin_2)`

Fournit un itérateur sur le dernier élément de l'intervalle [début_1, fin_1) tel que les éléments de la séquence débutant en début_1 soit égaux (au sens de ==) aux éléments de l'intervalle [début_2, fin_2). Si un tel élément n'existe pas, fournit la valeur fin_1 (attention, il ne s'agit pas nécessairement de end(). Complexité : au maximum (N1- N2 + 1) * N2 comparaisons.

`lu find_end (lu début_1, lu fin_1, lu début_2, lu fin_2, prédicat_b)`

Fonctionne comme la version précédente, avec cette différence que la comparaison d'égalité est remplacée par l'application du prédicat binaire prédicat_b. Complexité : au maximum (N1-N2 + 1) * N2 appels du prédicat.

FIND_FIRST_OF

lu find_first_of (lu début_1, lu fin_1, lu début_2, lu fin_2)

Recherche, dans l'intervalle [début_1, fin_1), le premier élément égal (au sens de ==) à l'un des éléments de l'intervalle [début_2, fin_2). Fournit un itérateur sur cet élément s'il existe, la valeur de fin_1, dans le cas contraire. Complexité : au maximum N1 * N2 comparaisons.

lu find_first_of (lu début_1, lu fin_1, lu début_2, lu fin_2, prédicat_b)

Recherche, dans l'intervalle [début_1, fin_1), le premier élément satisfaisant, avec l'un des éléments de l'intervalle [début_2, fin_2) au prédicat binaire prédicat_b. Fournit un itérateur sur cet élément s'il existe, la valeur de fin_1, dans le cas contraire. Complexité : au maximum N1 * N2 appels du prédicat

ADJACENT_FIND

lu adjacent_find (lu début, lu fin)

Recherche, dans l'intervalle [début, fin), la première occurrence de deux éléments successifs égaux (==) ; fournit un itérateur sur le premier des deux éléments égaux, s'ils existent, la valeur fin sinon.

lu adjacent_find (lu début, lu fin, prédicat_b)

Recherche, dans l'intervalle [début, fin), la première occurrence de deux éléments successifs satisfaisant au prédicat binaire prédicat_b ; fournit un itérateur sur le premier des deux éléments, s'ils existent, la valeur fin sinon.

SEARCH **lu search (lu début_1, lu fin_1, lu début_2, lu fin_2)**

Recherche, dans l'intervalle [début_1, fin_1), la première occurrence d'une séquence d'éléments identique (==) à celle de l'intervalle [début_2, fin_2). Fournit un itérateur sur le premier élément si cette occurrence, si elle existe, la fin fin_1 sinon. Complexité : au maximum N1 * N2 comparaisons.

lu search (lu début_1, lu fin_1, lu début_2, lu fin_2, prédicat_b)

Fonctionne comme la version précédente de search, avec cette différence que la comparaison de deux éléments de chacune des deux séquences se fait par le prédicat binaire prédicat_b, au lieu de se faire par égalité. Complexité : au maximum N1 * N2 appels du prédicat.

SEARCH_N **lu search_n (ludébut, lu fin, NbFois, valeur)**

Recherche dans l'intervalle [début, fin), une séquence de NbFois éléments égaux (au sens de ==) à valeur. Fournit un itérateur sur le premier élément si une telle séquence existe, la valeur fin sinon. Complexité : au maximum N comparaisons.

lu search_n (ludébut, lu fin, NbFois, valeur, prédicat_b)

Fonctionne comme la version précédente avec cette différence que la comparaison entre un élément et valeur se fait par le prédicat binaire prédicat_b, au lieu de se faire par égalité. Complexité : au maximum N applications du prédicat.

MAX_ELEMENT

lu max_element (lu début, lu fin)

Fournit un itérateur sur le premier élément de l'intervalle [début, fin) qui ne soit inférieur (<) à aucun des autres éléments de l'intervalle. Complexité : exactement N-1 comparaisons.

lu max_element (lu début, lu fin, prédicat_b)

Fonctionne comme la version précédente de max_element, mais en utilisant le prédicat binaire prédicat_b en lieu et place de l'opérateur <. Complexité : exactement N-1 appels du prédicat.

MIN_ELEMENT

lu min_element (lu début, lu fin)

Fournit un itérateur sur le premier élément de l'intervalle [début, fin) tel qu'aucun des autres éléments de l'intervalle ne lui soit inférieur (<). Complexité : exactement N-1 comparaisons.

lu min_element (lu début, lu fin, prédicat_b)

Fonctionne comme la version précédente de min_element, mais en utilisant le prédicat binaire prédicat_b en lieu et place de l'opérateur <. Complexité : exactement N-1 appels du prédicat.

3. ALGORITHMES DE TRANSFORMATION D'UNE SEQUENCE

REVERSE **void reverse (lb début, lb fin)**

Inverse le contenu de l'intervalle [début, fin). Complexité exactement N/2 échanges.

REVERSE_COPY **ls reverse_copy (lb début, lb fin, ls position)**

Copie l'intervalle [début, fin), dans l'ordre inverse, à partir de position ; les emplacements correspondants doivent exister ; attention, ici position désigne donc l'emplacement de la première copie et aussi le début de l'intervalle ; renvoie un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher. Complexité : exactement N affectations.

REPLACE **void replace (lu début, lu fin, anc_valeur, nouv_valeur)**

Remplace, dans l'intervalle [début, fin), tous les éléments égaux (==) à anc_valeur par nouv_valeur. Complexité : exactement N comparaisons.

REPLACE_IF **void replace_if (lu début, lu fin, prédicat_u, nouv_valeur)**

Remplace, dans l'intervalle [début, fin), tous les éléments satisfaisant au prédicat unaire prédicat_u par nouv_valeur. Complexité : exactement N applications du prédicat.

REPLACE_COPY

ls replace_copy (le début, le fin, ls position, anc_valeur, nouv_valeur)

Recopie l'intervalle [début, fin) à partir de position, en remplaçant tous les éléments égaux (==) à anc_valeur par nouv_valeur ; les emplacements correspondants doivent exister. Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher. Complexité : exactement N comparaisons.

REPLACE_COPY_IF

ls replace_copy_if (le début, le fin, ls position, prédicat_u, nouv_valeur)

Recopie l'intervalle [début, fin) à partir de position, en remplaçant tous les éléments satisfaisant au prédicat unaire prédicat_u par nouv_valeur ; les emplacements correspondants doivent exister Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher. Complexité : exactement N applications du prédicat.

ROTATE void rotate (lu début, lu milieu, lu fin)

Effectue une permutation circulaire (vers la gauche) des éléments de l'intervalle [début, fin) dont l'ampleur est telle que, après permutation, l'élément désigné par milieu soit venu en début. Complexité : au maximum N échanges.

ROTATE_COPY ls rotate_copy (lu début, lu milieu, lu fin, ls position)

Recopie, à partir de position, les éléments de l'intervalle [début, fin), affectés d'une permutation circulaire définie de la même façon que pour rotate ; les emplacements correspondants doivent exister. Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Complexité : au maximum N affectations.

PARTITION lb partition (lb début, lb fin, Prédicat_u)

Effectue une partition de l'intervalle [début, fin) en se fondant sur le prédicat unaire prédicat_u ; il s'agit d'une réorganisation telle que tous les éléments satisfaisant au prédicat arrivent avant tous les autres. Fournit un itérateur it tel que les éléments de l'intervalle [début, it) satisfont au prédicat, tandis que les éléments de l'intervalle [it, fin) n'y satisfont pas. Complexité : au maximum N/2 échanges et exactement N appels du prédicat.

STABLE_PARTITION lb stable_partition (lb début, lb fin, Prédicat_u)

Fonctionne comme partition, avec cette différence que les positions relatives des différents éléments à l'intérieur de chacune des deux parties soient préservées. Complexité : exactement N appels du prédicat et au maximum N Log N échanges (et même k N si l'on dispose de suffisamment de mémoire).

NEXT_PERMUTATION

bool next_permutation (lb début, lb fin)

Cet algorithme réalise ce que l'on nomme la "permutation suivante" des éléments de l'intervalle [début, fin). Il suppose que l'ensemble des permutations possibles est ordonné à partir de l'opérateur <, d'une manière lexicographique. On considère que la permutation suivant la dernière possible n'est rien d'autre que la première. Fournit la valeur true s'il existait bien une permutation suivante et la valeur false dans le cas où l'on est revenu à la première permutation possible. Complexité : au maximum N/2 échanges.

bool next_permutation (lb début, lb fin, prédicat_b)

Fonctionne comme la version précédente, avec cette seule différence que l'ensemble des permutations possibles et ordonné à partir du prédicat binaire prédicat_b. Complexité : au maximum N/2 échanges.

PREV_PERMUTATION

bool prev_permutation (lb début, lb fin)

bool prev_permutation (lb début, lb fin, prédicat_b)

Ces deux algorithmes fonctionnent comme next_permutation, en inversant simplement l'ordre des permutations possibles.

RANDOM_SHUFFLE

void random_shuffle (la début, la fin)

Répartit au hasard les éléments de l'intervalle [début, fin). Complexité : exactement N-1 échanges.

void random_shuffle (la début, la fin, générateur)

Même chose que random_shuffle, mais en utilisant la fonction générateur pour générer des nombres au hasard. Cette fonction doit fournir une valeur appartenant à l'intervalle [0, n), n étant une valeur fournie en argument. Complexité : exactement N-1 échanges.

TRANSFORM

ls transform (le début, le fin, ls position, opération_u)

Place à partir de position (les éléments correspondants doivent exister) les valeurs obtenues en appliquant la fonction unaire (à un argument) opération_u à chacune des valeurs de l'intervalle [début, fin). Fournit un itérateur sur la fin de l'intervalle ainsi rempli.

ls transform (le début_1, le fin_1, le début_2, ls position, opération_b)

Place à partir de position (les éléments correspondants doivent exister) les valeurs obtenues en appliquant la fonction binaire (à deux arguments) opération_b à chacune des valeurs de même rang de l'intervalle [début_1, fin_1) et de l'intervalle de même taille commençant en début_2. Fournit un itérateur sur la fin de l'intervalle ainsi rempli.

4. ALGORITHMES DE SUPPRESSION

REMOVE **lu remove (lu début, lu fin, valeur)**

Fournit un itérateur it tel que l'intervalle [début, it) contienne toutes les valeurs initialement présentes dans l'intervalle [début, fin), débarrassées de celles qui sont égales (==) à valeur. Attention, aucun élément n'est détruit ; tout au plus, peut-il avoir changé de valeur. L'algorithme est stable, c'est-à-dire que les valeurs non éliminées conservent leur ordre relatif. Complexité : exactement N comparaisons.

REMOVE_IF **lu remove_if (lu début, lu fin, prédicat_u)**

Fonctionne comme remove, avec cette différence que la condition d'élimination est fournie sous forme d'un prédicat unaire prédicat_u. Complexité : exactement N appels du prédicat.

REMOVE_COPY **ls remove_copy (le début, le fin, ls position, valeur)**

Recopie l'intervalle [début, fin) à partir de position (les éléments correspondants doivent exister), en supprimant les éléments égaux (==) à valeur. Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher. Comme remove, l'algorithme est stable. Complexité : exactement N comparaisons.

REMOVE_COPY_IF **ls remove_if (le début, le fin, ls position, prédicat_u)**

Fonctionne comme remove_copy, avec cette différence que la condition d'élimination est fournie sous forme d'un prédicat unaire prédicat_u. Complexité : exactement N appels du prédicat.

UNIQUE **lu unique (lu début, lu fin)**

Fournit un itérateur it tel que l'intervalle [début, it) corresponde à l'intervalle [début, fin), dans lequel les séquences de plusieurs valeurs consécutives égales (==) sont remplacées par la première. Attention, aucun élément n'est détruit ; tout au plus, peut-il avoir changé de place et de valeur. Complexité : exactement N comparaisons.

lu unique (lu début, lu fin, prédicat_b)

Fonctionne comme la version précédente, avec cette différence que la condition de répétition est fournie sous forme d'un prédicat binaire prédicat_b. Complexité : exactement N appels du prédicat.

UNIQUE_COPY

ls unique_copy (le début, le fin, ls position)

Recopie l'intervalle [début, fin) à partir de position (les éléments correspondants doivent exister), en ne conservant que la première valeur des séquences de plusieurs valeurs consécutives égales (==). Fournit un itérateur sur la fin de l'intervalle où s'est faite la copie. Les deux intervalles ne doivent pas se chevaucher. Complexité : exactement N comparaisons.

ls unique_copy (le début, le fin, ls position, prédicat_b)

Fonctionne comme unique_copy, avec cette différence que la condition de répétition de deux valeurs est fournie sous forme d'un prédicat binaire prédicat_u. On notera que la décision d'élimination d'une valeur se fait toujours par comparaison avec la précédente et non avec la première d'une séquence ; cette remarque n'a en fait d'importance qu'au cas où le prédicat fourni ne serait pas transitif... Complexité : exactement N appels du prédicat.

5. ALGORITHMES DE TRI

SORT

void sort (la début, la fin)

Trie les éléments de l'intervalle [début, fin), en se fondant sur l'opérateur <. L'algorithme n'est pas stable, c'est-à-dire que l'ordre relatif des éléments équivalents (au sens de <) n'est pas nécessairement respecté. Complexité : en moyenne N Log N comparaisons.

void sort (la début, la fin, fct_comp)

Trie les éléments de l'intervalle [début, fin), en se fondant sur le prédicat binaire fct_comp. Complexité : en moyenne N Log N appels du prédicat.

STABLE_SORT

void stable_sort (la début, la fin)

Trie les éléments de l'intervalle [début, fin), en se basant sur l'opérateur <. Contrairement à sort, cet algorithme est stable. Complexité : au maximum N $(\text{Log } N)^2$ comparaisons ; si l'implémentation dispose d'assez de mémoire, on peut descendre à N Log N comparaisons.

void stable_sort (la début, la fin, fct_comp)

Même chose que stable_sort en se basant sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict. Complexité : au maximum N $(\text{Log } N)^2$ applications du prédicat ; si l'implémentation dispose d'assez de mémoire, on peut descendre à N Log N appels.

PARTIAL_SORT

void partial_sort (la début, la milieu, la fin)

Réalise un tri partiel des éléments de l'intervalle [début, fin), en se basant sur l'opérateur < et en plaçant les premiers éléments convenablement triés dans l'intervalle [début, milieu) (c'est la taille de cet intervalle qui définit l'ampleur du tri). Les éléments de l'intervalle [milieu, fin) sont placés dans un ordre quelconque. Aucune contrainte de stabilité n'est imposée. Complexité : environ N Log N' comparaisons, N' étant le nombre d'éléments triés.

void partial_sort (la début, la milieu, la fin, fct_comp)

Fonctionne comme partial_sort, avec cette différence qu'au lieu de se fonder sur l'opérateur <, cet algorithme se fonde sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict. Complexité : environ N Log N' comparaisons, N' étant le nombre d'éléments triés.

PARTIAL_SORT_COPY

la partial_sort_copy (le début, le fin, la pos_début, la pos_fin)

Place dans l'intervalle [pos_début, pos_fin) le résultat du tri partiel ou total des éléments de l'intervalle [début, fin). Si l'intervalle de destination comporte plus d'éléments que l'intervalle de départ, ses derniers éléments ne seront pas utilisés. Fournit un itérateur sur la fin de l'intervalle de destination (pos_fin lorsque ce dernier est de taille inférieure ou égale à l'intervalle d'origine). Les deux intervalles ne doivent pas se chevaucher. Complexité : environ N Log N' comparaisons, N' étant le nombre d'éléments effectivement triés.

la partial_sort_copy (le début, le fin, la pos_début, la pos_fin, fct_comp)

Fonctionne comme partial_sort_copy avec cette différence qu'au lieu de se fonder sur l'opérateur <, cet algorithme se fonde sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict. Complexité : environ N Log N' comparaisons, N' étant le nombre d'éléments triés.

NTH_ELEMENT

void nth_element (la début, la position, la fin)

Place dans l'emplacement désigné par position – qui doit donc appartenir à l'intervalle [début, fin) – l'élément de l'intervalle [début, fin) qui se trouverait là, à la suite d'un tri. Les autres éléments de l'intervalle peuvent changer de place. Complexité : en moyenne N comparaisons.

void nth_element (la début, la position, la fin, fct_comp)

Fonctionne comme la version précédente, avec cette différence qu'au lieu de se fonder sur l'opérateur <, cet algorithme se fonde sur le prédicat binaire fct_comp qui doit correspondre à une relation d'ordre faible strict. Complexité : en moyenne N applications du prédicat.

6. ALGORITHMES DE RECHERCHE ET DE FUSIONS SUR DES SEQUENCES ORDONNEES

N.B. Tous ces algorithmes peuvent fonctionner avec de simples itérateurs unidirectionnels. Mais, lorsque l'on dispose d'itérateurs à accès direct, on peut augmenter légèrement les performances, dans la mesure où certaines séries de p incrémentations de la forme it++ peuvent être remplacées par une seule it+=p ; plus précisément, on passe de O(N) à O(Log N) incrémentations.

LOWER_BOUND

lu lower_bound (lu début, lu fin, valeur)

Fournit un itérateur sur la première position où valeur peut être insérée, compte tenu de l'ordre induit par l'opérateur <. Complexité : au maximum Log N+1 comparaisons.

lu lower_bound (lu début, lu fin, valeur, fct_comp)

Fournit un itérateur sur la première position où valeur peut être insérée, compte tenu de l'ordre induit par le prédicat binaire fct_comp. Complexité : au maximum Log N+1 comparaisons.

UPPER_BOUND

lu upper_bound (lu début, lu fin, valeur)

Fournit un itérateur sur la dernière position où valeur peut être insérée, compte tenu de l'ordre induit par l'opérateur <. Complexité : au maximum Log N+1 comparaisons.

lu upper_bound (lu début, lu fin, valeur, fct_comp)

Fournit un itérateur sur la dernière position où valeur peut être insérée, compte tenu de l'ordre induit par le prédicat binaire fct_comp. Complexité : au maximum Log N+1 comparaisons.

EQUAL_RANGE

pair < lu, lu> equal_range (lu début, lu fin, valeur)

Fournit le plus grand intervalle [it1, it2) tel que valeur puisse être insérée en n'importe quel point de cet intervalle, compte tenu de l'ordre induit par l'opérateur <. Complexité : au maximum 2 Log N+1 comparaisons.

pair < lu, lu> equal_range (lu début, lu fin, valeur, fct_comp)

Fonctionne comme la version précédente, en se basant sur l'ordre induit par le prédicat binaire fct_comp au lieu de l'opérateur <.

BINARY_SEARCH

bool binary_search (lu début, lu fin, valeur)

Fournit la valeur true s'il existe, dans l'intervalle [début, fin), un élément équivalent à valeur, et la valeur false, dans le cas contraire. Complexité : au plus Log N+2 comparaisons.

bool binary_search (lu début, lu fin, valeur, fct_comp)

Fournit la valeur true s'il existe, dans l'intervalle [début, fin), un élément équivalent à valeur (au sens de la relation induite par le prédicat fct_comp) et la valeur false dans le cas contraire. Complexité : au plus Log N+2 appels du prédicat.

MERGE

ls merge (le début_1, le fin_1, le début_2, le fin_2, ls position)

Fusionne les deux intervalles [début_1, fin_1) et [début_2, fin_2), à partir de position (les éléments correspondants doivent exister), en se fondant sur l'ordre induit par l'opérateur <. L'algorithme est stable : l'ordre relatif d'éléments équivalents dans l'un des intervalles d'origine est respecté dans l'intervalle d'arrivée ; si des éléments équivalents apparaissent dans les intervalles à fusionner, ceux du premier intervalle apparaissent toujours avant ceux du second. L'intervalle d'arrivée ne doit pas se chevaucher avec les intervalles d'origine (en revanche, rien n'interdit que les deux intervalles d'origine se chevauchent). Complexité : au plus N1+N2-1 comparaisons.

ls merge (le début_1, le fin_1, le début_2, le fin_2, ls position, fct_comp)

Fonctionne comme la version précédente, avec cette différence que l'on se base sur l'ordre induit par le prédicat binaire fct_comp. Complexité : au plus N1+N2-1 appels du prédicat.

INPLACE_MERGE

void inplace_merge (lb début, lb milieu, lb fin)

Fusionne les deux intervalles [début, milieu) et [milieu, fin) dans l'intervalle [début, fin) en se basant sur l'ordre induit par l'opérateur <. Complexité : N-1 comparaisons si l'on dispose de suffisamment de mémoire, N Log N comparaisons sinon.

void inplace_merge (lb début, lb milieu, lb fin, fct_comp)

Fonctionne comme la version précédente, avec cette différence que l'on se base sur l'ordre induit par le prédicat binaire fct_comp. Complexité : N-1 appels du prédicat, si l'on dispose de suffisamment de mémoire, N Log N appels sinon.

7. ALGORITHMES A CARACTERE NUMERIQUE

ACCUMULATE

valeur accumulate (le debut, le fin, val_init)

Fournit la valeur obtenue en ajoutant (opérateur +) à la valeur initiale val_init, la valeur de chacun des éléments de l'intervalle [début, fin).

valeur accumulate (le debut, le fin, val_initiale, fct_cumul)

Fonctionne comme la version précédente, en la généralisant : l'opération appliquée n'étant plus définie par l'opérateur +, mais par la fonction fct_cumul, recevant deux arguments du type des éléments concernés et fournissant un résultat de ce même type (la valeur accumulée courante est fournie en premier argument, celle de l'élément courant, en second).

INNER_PRODUCT

valeur inner_product (le début_1, le fin_1, le début_2, val_init)

Fournit le produit scalaire de la séquence des valeurs de l'intervalle [début_1, fin_2) et de la séquence de valeurs de même longueur débutant en début_2, augmenté de la valeur initiale val_init.

valeur inner_product (le début_1, le fin_1, le début_2, val_init, fct_cumul, fct_prod)

Fonctionne comme la version précédente, en remplaçant l'opération de cumul (+) par l'appel de la fonction fct_cumul (la valeur cumulée est fournie en premier argument) et l'opération de produit par l'appel de la fonction fct_prod (la valeur courante du premier intervalle étant fournie en premier argument).

PARTIAL_SUM

ls partial_sum (le début, le fin, ls position)

Crée, à partir de position (les éléments correspondants doivent exister), un intervalle de même taille que l'intervalle [début, fin), contenant les sommes partielles du premier intervalle : le premier élément correspond à la première valeur de [début, fin), le second élément à la somme des deux premières valeurs et ainsi de suite. Fournit un itérateur sur la fin de l'intervalle créé.

ls partial_sum (le début, le fin, ls position, fct_cumul)

Fonctionne comme la version précédente, en remplaçant l'opération de sommation (+) par l'appel de la fonction fct_cumul (la valeur cumulée est fournie en premier argument).

ADJACENT_DIFFERENCE

ls ajacent_difference (le début, le fin, ls position)

Crée, à partir de position (les éléments correspondants doivent exister), un intervalle de même taille que l'intervalle [début, fin), contenant les différences entre deux éléments consécutifs de ce premier intervalle : l'élément de rang i, hormis le premier, s'obtient en faisant la différence (opérateur -) entre l'élément de rang i et celui de rang i-1. Le premier élément reste inchangé. Fournit un itérateur sur la fin de l'intervalle créé.

ls ajacent_difference (le début, le fin, ls position, fct_diff)

Fonctionne comme la version précédente, en remplaçant l'opération de différence (-) par l'appel de la fonction fct_diff.

8. ALGORITHMES A CARACTERE ENSEMBLISTE

INCLUDES

bool includes (le début_1, le fin_1, le début_2, le fin_2)

Fournit la valeur true si, à toute valeur appartenant à l'intervalle [début_1, fin_1), correspond une valeur égale (==) dans l'intervalle [début_2, fin_2), avec la même pluralité : autrement dit, (si une valeur figure n fois dans le premier intervalle, elle devra figurer au moins n fois dans le second intervalle). Complexité : au maximum 2 N1*N2-1 comparaisons.

bool includes (le début_1, le fin_1, le début_2, le fin_2, fct_comp)

Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Complexité : au maximum 2 N1*N2-1 appels du prédicat

SET_UNION

ls set_union (le début_1, le fin_1, le début_2, le fin_2, ls position)

Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant au moins à l'un des deux intervalles [début_1, fin_1) [début_2, fin_2), avec la pluralité maximale : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra max(n, n') fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments (==) devra correspondre aux classes d'équivalence de R. Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé. Complexité : au maximum 2*N1*N2-1 comparaisons.

ls set_union (le début_1, le fin_1, le début_2, le fin_2, ls position, fct_comp)

Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles. Complexité : au maximum 2*N1*N2-1 appels du prédicat.

SET_INTERSECTION

ls set_intersection (le début_1, le fin_1, le début_2, le fin_2, ls position)

Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant simultanément aux deux intervalles [début_1, fin_1) [début_2, fin_2), avec la pluralité minimale : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra min(n, n') fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments (==) devra correspondre aux classes d'équivalence de R. Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé. Complexité : au maximum 2*N1*N2-1 comparaisons.

ls set_intersection (le début_1, le fin_1, le début_2, le fin_2, ls position, fct_comp)

Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles. Complexité : au maximum 2*N1*N2-1 appels du prédicat.

SET_DIFFERENCE

ls set_difference (le début_1, le fin_1, le début_2, le fin_2, ls position)

Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant à l'intervalle [début_1, fin_1) sans appartenir à l'intervalle [début_2, fin_2) ; on tient compte de la pluralité : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra max(0, n-n') fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments (==) devra correspondre aux classes d'équivalence de R. Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé. Complexité : au maximum 2*N1*N2-1 comparaisons.

ls set_difference (le début_1, le fin_1, le début_2, le fin_2, ls position, fct_comp)

Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles. Complexité : au maximum 2*N1*N2-1 appels du prédicat.

SET_SYMMETRIC_DIFFERENCE

ls set_symetric_difference (le début_1, le fin_1, le début_2, le fin_2, ls position)

Crée, à partir de position (les éléments correspondants doivent exister), une séquence formée des éléments appartenant à l'intervalle [début_1, fin_1) sans appartenir à l'intervalle [début_2, fin_2) ou appartenant au second, sans appartenir au premir ; on tient compte de la pluralité : si un élément apparaît n fois dans le premier intervalle et n' fois dans le second, il apparaîtra |n-n'| fois dans le résultat. Les éléments doivent être triés suivant la même relation R et l'égalité de deux éléments (==) devra correspondre aux classes d'équivalence de R. Les deux intervalles ne doivent pas se chevaucher. Fournit un itérateur sur la fin de l'intervalle créé. Complexité : au maximum 2*N1*N2-1 comparaisons.

ls set_symetric_difference (le début_1, le fin_1, le début_2, le fin_2, ls position, fct_comp)

Fonctionne comme la version précédente, mais en utilisant le prédicat binaire fct_comp pour décider de l'égalité de deux valeurs. Là encore, ce dernier doit correspondre aux classes d'équivalence de la relation ayant servi à ordonner les deux intervalles. Complexité : au maximum 2*N1*N2-1 appels du prédicat.

9. ALGORITHMES DIVERS

COUNT

nombre count (le début, le fin, valeur)

Fournit le nombre de valeurs de l'intervalle [début, fin) égales à valeur (au sens de ==).

nombre count (le début, le fin, prédicat_u)

Fournit le nombre de valeurs de l'intervalle [début, fin) satisfaisant au prédicat unaire prédicat_u.

FOR_EACH **fct for_each (le début, le fin, fct)**

Applique la fonction fct à chacun des éléments de l'intervalle [début, fin) ; fournit fct en résultat.

EQUAL **bool equal (le début_1, le fin_1, le début_2)**

Fournit la valeur true si tous les éléments de l'intervalle [début_1, fin_2) sont égaux (au sens de ==) aux éléments correspondants de l'intervalle de même taille commençant en début_2.

bool equal (le début_1, le fin_1, le début_2, prédicat_b)

Fonctionne comme la version précédente, en utilisant le prédicat binaire prédicat_b, à la place de l'opérateur ==.

ITER_SWAP **void iter_swap (lu pos1, lu pos2)**

Echange les valeurs des éléments désignés par les deux itérateurs pos1 et pos2.

CORRECTION DES EXERCICES

Nous vous fournissons ici la "correction" des exercices dont l'énoncé est précédé de l'indication (C). Bien entendu, les programmes proposés doivent être considérés comme une solution parmi (beaucoup) d'autres.

CHAPITRE V

Exercice 5.2

```
#include <iostream.h>
         /* déclaration de la classe vecteur */
class vecteur
{  double x, y, z ;
  public :
   void initialise (double, double, double) ;
   void homothetie (double) ;
   void affiche () ;
} ;
         /* définition des fonctions membre de la classe vecteur */
void vecteur::initialise (double a, double b, double c)
{ x = a ; y = b ; z = c ;
}
void vecteur::homothetie (double coeff)
{
  x = x * coeff ; y = y * coeff ; z = z * coeff ;
}
void vecteur::affiche ()
{
  cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
         /* programme de test de la classe vecteur */
main()
{ vecteur v1, v2 ;
  v1.initialise (1.0, 2.5, 5.8) ; v1.affiche () ;
  v2.initialise (12.5, 3.8, 0.0) ; v2.affiche () ;
  v1.homothetie (3.5) ; v1.affiche () ;
  v2 = v1 ; v2.affiche () ;
}
```

Exercice 5.3

```
#include <iostream.h>
         /* déclaration de la classe vecteur */
```

```
class vecteur
{  double x, y, z ;
  public :
   vecteur (double, double, double) ;  // constructeur
   void homothetie (double) ;
   void affiche () ;
} ;
         /* définition des fonctions membre de la classe vecteur */
vecteur::vecteur (double a, double b, double c)   // attention, pas de void ..
{
  x = a ; y = b ; z = c ;
}
void vecteur::homothetie (double coeff)
{ x = x * coeff ; y = y * coeff ; z = z * coeff ;
}
void vecteur::affiche ()
{ cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
         /* programme de test de la classe vecteur */
main()
{ vecteur v1(1.0, 2.5, 5.8) ;       // vecteur v1 serait ici invalide
  vecteur v2(12.5, 3.8, 0.0) ;
  v1.affiche () ;
  v2.affiche () ;
  v1.homothetie (3.5) ; v1.affiche () ;
  v2 = v1 ; v2.affiche () ;
}
```

CHAPITRE VI

Exercice 6.1

a) Avec des fonctions membre indépendantes

```
#include <iostream.h>
         /* déclaration de la classe vecteur */
class vecteur
{  double x, y, z ;
  public :
   vecteur () ;                          // constructeur 1
   vecteur (double, double, double) ;    // constructeur 2
   void affiche () ;
} ;
         /* définition des fonctions membre de la classe vecteur */
vecteur::vecteur ()
{
  x=0 ; y=0 ; z=0 ;
}
vecteur::vecteur (double a, double b, double c)
{
  x = a ; y = b ; z = c ;
}
void vecteur::affiche ()
{
```

```
  cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
        /* programme de test de la classe vecteur */
main()
{ vecteur v1 ;       // attention vecteur v1 () aurait une autre signification
                     // v1 serait une fonction sans argument, fournissant un
                     // résultat de type vecteur
  vecteur v2(12.5, 3.8, 0.0) ;
                     // ces déclarations seraient ici invalides :
                     // vecteur v3 (5) ; vecteur v4 (2.5, 4) ;
  v1.affiche () ;
  v2.affiche () ;
}
```

b) Avec des fonctions membre "en ligne"

```
#include <iostream.h>
        /* déclaration de la classe vecteur */
class vecteur
{
   double x, y, z ;
  public :
   vecteur ()                                    // constructeur 1
    { x=0 ; y=0 ; z=0 ; }
   vecteur (double a, double b, double c)        // constructeur 2
    { x=a ; y=b ; z=c ; }
   void affiche ()
    { cout << "Vecteur de coordonnées : "
        << x << " " << y << " " << z << "\n" ;
    }
} ;

        /* programme de test de la classe vecteur */
main()
{
  vecteur v1, v2(3,4,5) ;
  v1.affiche () ;
  v2.affiche () ;
}
```

Exercice 6.2

```
#include <iostream.h>
        /* déclaration de la classe vecteur */
class vecteur
{
   double x, y, z ;
  public :
   vecteur () ;                          // constructeur 1
   vecteur (double, double, double) ;    // constructeur 2
   void affiche () ;
```

```
    int prod_scal (vecteur) ;
} ;

        /* définition des fonctions membre de la classe vecteur */
vecteur::vecteur ()
{
  x=0 ; y=0 ; z=0 ;
}
vecteur::vecteur (double a, double b, double c)
{
  x = a ; y = b ; z = c ;
}
void vecteur::affiche ()
{
  cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
int vecteur::prod_scal (vecteur v)
{
  return (x * v.x + y * v.y + z * v.z) ;
}
main()   /* programme de test de la classe vecteur */
{
  vecteur v1 (1,2,3) ;
  vecteur v2 (5,4,3) ;
  v1.affiche () ; v2.affiche () ;
  int ps ;
  ps = v1.prod_scal (v2) ; cout << "V1.V2 = " << ps << "\n" ;
  ps = v2.prod_scal (v1) ; cout << "V2.V1 = " << ps << "\n" ;
  cout << "V1.V1 = " << v1.prod_scal (v1) << "\n" ;
  cout << "V2.V2 = " << v2.prod_scal (v2) << "\n" ;
}
```

Exercice 6.3

```
#include <iostream.h>
        /* déclaration de la classe vecteur */
class vecteur
{  double x, y, z ;
  public :
   vecteur () ;                          // constructeur 1
   vecteur (double, double, double) ;    // constructeur 2
   void affiche () ;
   int prod_scal (vecteur) ;
   vecteur somme (vecteur) ;
} ;
        /* définition des fonctions membre de la classe vecteur */
vecteur::vecteur ()
{ x=0 ; y=0 ; z=0 ;
}
vecteur::vecteur (double a, double b, double c)
{ x = a ; y = b ; z = c ;
}
void vecteur::affiche ()
{ cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
```

```
int vecteur::prod_scal (vecteur v)
{ return (x * v.x + y * v.y + z * v.z) ;
}
vecteur vecteur::somme (vecteur v)
{ vecteur res ;
  res.x = x + v.x ; res.y = y + v.y ; res.z = z + v.z ;
  return res ;
}
         /* programme de test de la classe vecteur */
main()
{ vecteur v1 (1,2,3) ;
  vecteur v2 (5,4,3) ;
  vecteur v3 ;
  v1.affiche () ; v2.affiche () ; v3.affiche () ;
  v3 = v1.somme (v2) ; v3.affiche () ;
  v3 = v2.somme (v1) ; v3.affiche () ;
}
```

Exercice 6.4

a) Transmission par adresse des valeurs de type vecteur

```
         /* déclaration de la classe vecteur */
class vecteur
{  double x, y, z ;
  public :
   vecteur () ;                          // constructeur 1
   vecteur (double, double, double) ;    // constructeur 2
   void affiche () ;
   int prod_scal (vecteur *) ;
   vecteur somme (vecteur *) ;
} ;
         /* définition des fonctions membre de la classe vecteur */
vecteur::vecteur ()
{
  x=0 ; y=0 ; z=0 ;
}
vecteur::vecteur (double a, double b, double c)
{
  x = a ; y = b ; z = c ;
}
void vecteur::affiche ()
{
  cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
int vecteur::prod_scal (vecteur * adv)
{
  return (x * adv->x + y * adv->y + z * adv->z) ;
       // on pourrait écrire, de façon plus symétrique :
       //  return (this->x * adv->x + this->y * adv->y + this->z * adv-> z ; }
}
vecteur vecteur::somme (vecteur * adv)
{
  vecteur res ;
  res.x = x + adv->x ; res.y = y + adv->y ; res.z = z + adv->z ;
```

```
        // ou, pour conserver la symétrie :
        // res.x = this->x + adv-> x ; .......
  return res ;
        // attention, on ne peut pas transmettre l'adresse de res, car
        // il s'agit d'une variable automatique
}
         /* programme de test de la classe vecteur */
main()
{
  vecteur v1 (1,2,3) ;
  vecteur v2 (5,4,3) ;
  vecteur v3 ;
  v1.affiche () ; v2.affiche () ; v3.affiche () ;
  v3 = v1.somme (&v2) ; v3.affiche () ;
  v3 = v2.somme (&v1) ; v3.affiche () ;
}
```

b) Transmission par référence des valeurs de type vecteur

```
#include <iostream.h>
         /* déclaration de la classe vecteur */
class vecteur
{
   double x, y, z ;
  public :
   vecteur () ;                          // constructeur 1
   vecteur (double, double, double) ;    // constructeur 2
   void affiche () ;
   int prod_scal (vecteur &) ;
   vecteur somme (vecteur &) ;
} ;

         /* définition des fonctions membre de la classe vecteur */
vecteur::vecteur ()
{
  x=0 ; y=0 ; z=0 ;
}
vecteur::vecteur (double a, double b, double c)
{
  x = a ; y = b ; z = c ;
}
void vecteur::affiche ()
{
  cout << "Vecteur de coordonnées : " << x << " " << y << " " << z << "\n" ;
}
int vecteur::prod_scal (vecteur & v)
{
  return (x * v.x + y * v.y + z * v.z) ;
}
vecteur vecteur::somme (vecteur & v)
{
  vecteur res ;
  res.x = x + v.x ; res.y = y + v.y ; res.z = z + v.z ;
  return res ;
```

```
        // attention, on ne peut pas transmettre l'adresse de res, car
        // il s'agit d'une variable automatique
}


         /* programme de test de la classe vecteur */
main()
{
  vecteur v1 (1,2,3) ;
  vecteur v2 (5,4,3) ;
  vecteur v3 ;
  v1.affiche () ; v2.affiche () ; v3.affiche () ;
  v3 = v1.somme (v2) ; v3.affiche () ;
  v3 = v2.somme (v1) ; v3.affiche () ;
}
```

CHAPITRE VII

Exercice 7.5

```
#include <iostream.h>
       /* déclaration (et définition) de la classe pile_entier */
           /* ici, toutes les fonctions membre sont "inline" */
const Max = 20 ;
class pile_entier
{  int dim ;           // nombre maximal d'entiers de la pile
   int * adr ;         // adresse emplacement des dim entiers
   int nelem ;         // nombre d'entiers actuellement empilés
 public :
   pile_entier (int n = Max)         // constructeur(s)
     { adr = new int [dim=n] ;
       nelem = 0 ;
     }
   ~pile_entier ()                   // destructeur
     { delete adr ;
     }
   void empile (int p)
     { if (nelem < dim) adr[nelem++] = p ; }
   int depile ()
     { if (nelem > 0) return adr[--nelem] ;
                 else return 0 ;    // faute de mieux !
     }
   int pleine ()
     { return (nelem == dim) ; }
   int vide ()
     { return (nelem == 0 ) ; }
} ;
```

Exercice 7.6

```
             /* programme d'essai de la classe pile_entier */
main()
{
   int i ;
```

```
        /* exemples d'utilisation de piles automatiques */
    pile_entier a(3),        // une pile de 3 entiers
                b ;          // une pile de 20 entiers (par défaut)
    cout << "a pleine ? " << a.pleine () << "\n" ;
    cout << "a vide   ? " << a.vide ()   << "\n" ;
    a.empile (3) ; a.empile (9) ; a.empile (11) ;
    cout << "Contenu de a : " ;
    for (i=0 ; i<3 ; i++) cout << a.depile () << " " ;
    cout << "\n" ;
    for (i=0 ; i<30 ; i++)  b.empile (10*i) ;
    cout << "Contenu de b : " ;
    for (i=0 ; i<30 ; i++)  if ( ! b.vide() ) cout << b.depile () << " " ;
    cout << "\n" ;

        /* exemple d'utilisation d'une pile dynamique */
    pile_entier * adp = new pile_entier (5) ;
                        // pointeur sur une pile de 5 entiers
    cout << "pile dynamique vide ? " << adp->vide () << "\n" ;
    for (i=0 ; i<10 ; i++) adp->empile (10*i) ;
    cout << "Contenu de la pile dynamique : " ;
    for (i=0 ; i<10 ; i++) if ( ! adp->vide() ) cout << adp->depile () << " " ;
}
```

Exercice 7.8

```
#include <iostream.h>
       /* déclaration (et définition) de la classe pile_entiers */
const Max = 20 ;
class pile_entier
{
   int dim ;           // nombre maximal d'entiers de la pile
   int * adr ;         // adresse emplacement des dim entiers
   int nelem ;         // nombre d'entiers actuellement empilés

public :
   pile_entier (int n = Max)         // constructeur(s)
     { adr = new int [dim=n] ;
       nelem = 0 ;
     }
   ~pile_entier ()                   // destructeur
     { delete adr ;
     }
   void empile (int p)
     { if (nelem < dim) adr[nelem++] = p ; }
   int depile ()
     { if (nelem > 0) return adr[--nelem] ;
                 else return 0 ;    // faute de mieux !
     }
   int pleine ()
     { return (nelem == dim) ; }
   int vide ()
     { return (nelem == 0 ) ; }
   pile_entier (pile_entier &) ;             // constructeur de recopie
```

```
} ;
pile_entier::pile_entier (pile_entier & p)
{ adr = new int [dim = p.dim] ;
  nelem = p.nelem ;
  int i ;
  for (i=0 ; i<nelem ; i++) adr[i] = p.adr[i] ;
}

            /* programme d'essai de la classe pile_entier */
main()
{
   int i ;
   pile_entier a(3) ;                //  une pile a de 3 entiers
   a.empile (5) ; a.empile (12) ;
   pile_entier b = a ;               // une pile b égale à a
   cout << "Contenu de b : " ;
   for (i=0 ; i<3 ; i++)  if ( ! b.vide() ) cout << b.depile () << " " ;
   cout << "\n" ;
}
```

CHAPITRE IX

Exercice 9.3

a) Avec une fonction membre

```
#include <iostream.h>
     /* classe point avec surdéfinition de == comme fonction membre */
class point
{  int x, y ;
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   int operator == (point & p)  // on pourrait ne pas transmettre par référence
     { return ( (p.x == x) && (p.y == y) ) ; }
} ;

            /* programme de test de la classe point */
main()
{ point a(2,3), b(1), c(2,3) ;
  cout << " a == b   " << (a == b) << "\n" ;   // attention : parenthèses
  cout << " b == a   " << (b == a) << "\n" ;   // indispensables, compte tenu
  cout << " a == c   " << (a == c) << "\n" ;   // des priorités relatives de
  cout << " c == a   " << (c == a) << "\n" ;   // de == et de <<
}
```

b) Avec une fonction amie

```
#include <iostream.h>
     /* classe point avec surdéfinition de == comme fonction amie */
class point
{  int x, y ;
```

```
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   friend int operator == (point &, point &) ;
} ;
int operator == (point & p, point & q)
{ return ( (p.x == q.x) && (p.y == q.y) ) ;
}

            /* programme de test de la classe point */
main()
{ point a(2,3), b(1), c(2,3) ;
  cout << " a == b   " << (a == b) << "\n" ;   // attention : parenthèses
  cout << " b == a   " << (b == a) << "\n" ;   // indispensables, compte tenu
  cout << " a == c   " << (a == c) << "\n" ;   // des priorités relatives de
  cout << " c == a   " << (c == a) << "\n" ;   // de == et de <<
}
```

Exercice 9.4

a) Avec des fonctions membre

```
#include <iostream.h>
                /* la classe pile_entiers */
    /* avec surdéfinition des opérateurs < et > comme fonctions membre */
class pile_entier
{  int dim ;           // nombre maximal d'entiers de la pile
   int * adr ;         // adresse emplacement des dim entiers
   int nelem ;         // nombre d'entiers actuellement empilés
 public :
   pile_entier (int n)               // constructeur
     { adr = new int [dim=n] ;
       nelem = 0 ;
     }
   ~pile_entier ()                   // destructeur
     { delete adr ;
     }
   void operator < (int n)
     { if (nelem < dim) adr[nelem++] = n ;
     }
   void operator > (int & n )        // attention & indispensable ici
     { if (nelem > 0) n = adr[--nelem] ;
     }
} ;
            /* programme d'essai de la classe pile_entier */
main()
{
   int i, n ;
   pile_entier a(3) ;
   a < 3 ; a < 9 ; a < 11 ;
   cout << "Contenu de a : " ;
   for (i=0 ; i<3 ; i++)
      { a > n ; cout << n << " " ; }
   cout << "\n" ;
}
```

b) Avec des fonctions amies

```
#include <iostream.h>
                /* la classe pile_entiers */
    /* avec surdéfinition des opérateurs < et > comme fonctions amies */
class pile_entier
{  int dim ;            // nombre maximal d'entiers de la pile
   int * adr ;          // adresse emplacement des dim entiers
   int nelem ;          // nombre d'entiers actuellement empilés
 public :
   pile_entier (int n)               // constructeur
     { adr = new int [dim=n] ;
       nelem = 0 ;
     }
   ~pile_entier ()                   // destructeur
     { delete adr ;
     }
   friend void operator < (pile_entier &, int) ;   // & non indispensable
   friend void operator > (pile_entier &, int &) ; // int & indispensable ici
} ;
void operator < (pile_entier & p, int n)
{ if (p.nelem < p.dim) p.adr[p.nelem++] = n ;
}
void operator > (pile_entier & p, int & n)
{ if (p.nelem > 0) n = p.adr[--p.nelem] ;
}
            /* programme d'essai de la classe pile_entier */
main()
{  int i, n ;
   pile_entier a(3) ;
   a < 3 ; a < 9 ; a < 11 ;
   cout << "Contenu de a : " ;
   for (i=0 ; i<3 ; i++)
      { a > n ; cout << n << " " ; }
   cout << "\n" ;
}
```

Exercice 9.5

```
#include <iostream.h>
class chaine
{ int lg ;                // longueur actuelle de la chaîne
  char * adr ;            // adresse zone contenant la chaîne
 public :
  chaine () ;             // constructeur I
  chaine (char *) ;       // constructeur II
  chaine (chaine &) ;     // constructeur III (par recopie)
  ~chaine ()              // destructeur ("inline")
    { delete adr ; }
  void affiche () ;
  chaine & operator = (chaine &) ;
  int operator == (chaine &) ;
  chaine operator + (chaine &) ;
```

```
  char & operator [] (int) ;
} ;
chaine::chaine ()               // constructeur I
   {  lg = 0 ; adr=0 ;
   }
chaine::chaine (char * adc)     // constructeur II (à partir d'une chaîne C)
   { char * ad = adc ;
     lg = 0 ;
     while (*ad++) lg++ ;             // calcul longueur chaîne C
     adr = new char [lg] ;
     for (int i=0 ; i<lg ; i++)       // recopie chaîne C
        adr[i] = adc[i] ;
   }
chaine::chaine (chaine & ch)    // constructeur III (par recopie)
   { adr = new char [lg = ch.lg] ;
     for (int i=0; i<lg ; i++)
        adr[i] = ch.adr[i] ;
   }
void chaine::affiche ()
   { for (int i=0; i<lg; i++)
        cout << adr[i] ;                // en version < 2.0, utilisez printf
   }
chaine & chaine::operator = (chaine & ch)
   { if (this != & ch)                  // on ne fait rien pour a=a
       { delete adr ;
         adr = new char [lg = ch.lg] ;
         for (int i=0; i<lg ; i++)
             adr[i] = ch.adr[i] ;
       }
     return * this ;                    // pour pouvoir utiliser
   }                                    // la valeur de a=b
int chaine::operator == (chaine & ch)
   { for (int i=0 ; i<lg ; i++)
        if (adr[i] != ch.adr[i]) return 0 ;
     return 1 ;
   }
chaine chaine::operator + (chaine & ch)    // attention : la valeur de retour
   { chaine res ;                          // est à transmettre par valeur
     res.adr = new char [res.lg = lg + ch.lg] ;
     int i ;
     for (i=0 ; i<lg ; i++)    res.adr[i] = adr[i] ;
     for (i=0 ; i<ch.lg ; i++) res.adr[i+lg] = ch.adr[i] ;
     return res ;
   }
char & chaine::operator [] (int i)
   { return adr[i] ;                        // ici, on n'a pas prévu de
   }                                        // vérification de la valeur de i
main()
{ chaine a ;  cout << "chaine a : " ; a.affiche () ; cout << "\n" ;
  chaine b("bonjour") ; cout << "chaine b : " ; b.affiche () ; cout << "\n" ;
  chaine c=b ; cout << "chaine c : " ; c.affiche () ; cout << "\n" ;
  chaine d("hello") ;
  a = b = d ;
  cout << "chaine b : " ; b.affiche () ; cout << "\n" ;
  cout << "chaine a : " ; a.affiche () ; cout << "\n" ;
```

```
    cout << "a == b   : " << (a == b) << "\n" ;
    chaine x("salut "), y("chère "), z("madame");
    a = x + y + z ;
    cout << "chaine a : " ; a.affiche () ; cout << "\n" ;
    a = a ;
    cout << "chaine a : " ; a.affiche () ; cout << "\n" ;
    chaine e("xxxxxxxxxx") ;
    for (char cr='a', i=0 ; cr<'f' ; cr++, i++ ) e[i] = cr ;
    cout << "chaine e : " ; e.affiche () ; cout << "\n" ;
}
```

INDEX